San Diego, CA 92152-5000

2

Technical Document 552
October 1989

KAPSE Interface Team (KIT) Public Report

Volume VIII
(Part 2 of 2)

D. L. Hayward

Prepared for the Ada Joint Program Office

NAVAL OCEAN SYSTEMS CENTER

San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN	R. M. HILLYER
Commander	Technical Director

ADMINISTRATIVE INFORMATION

This work was performed by the Computer Systems Software and Technology Branch, Code 411, of the Naval Ocean Systems Center, San Diego, CA, for the Ada Joint Program Office, Pentagon, 1211 S. Fern Street, Washington, DC 20301-3081 and represents evolving ideas and progress of the KAPSE Interface Team (KIT).

Released by	Under authority of
R. A. Wasilausky, Head	A. G. Justice, Head
Computer Systems Software	Information Processing and
and Technology Branch	Displaying Division

FS

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
October 1989

3. REPORT TYPE AND DATES COVERED
Final May 1985 to October 1985

4. TITLE AND SUBTITLE
KAPSE INTERFACE TEAM (KIT) PUBLIC REPORT
Volume VIII (Part 2 of 2)

5. FUNDING NUMBERS
C:
PE: 0603226F
PR:
WU: DN288 534

6. AUTHOR(S)
D. L. Hayward

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Ocean Systems Center
San Diego, CA 92152-5000

8. PERFORMING ORGANIZATION REPORT NUMBER
NOSC TD 552

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
Pentagon
1211 S. Fern Street
Washington, DC 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report is the eighth in a series and represents evolving ideas and progress of the KAPSE Interface Team (KIT).

14. SUBJECT TERMS
software engineering
CAIS Ada
APSE interface standard
programming languages

15. NUMBER OF PAGES
393

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

Partial

CONTENTS

1. INTRODUCTION 1-1

Meetings 1-1

Program Management 1-2

Common APSE Interface Set (CAIS) Requirements and Design Criteria (RAC) 1-2

COMPWG 1-3

GACWG 1-3

DEFWG 1-3

Prototypes 1-3

Related Efforts 1-3

Conclusion 1-4

2. TEAM PROCEEDINGS 2-1

3. KIT/KITIA DOCUMENTATION 3-1

Ada Programming Support Environment Interoperability and Transportability Management Plan 3-1

CAIS 2 Progress Report 3-63

Common APSE Interface Set (CAIS) Version 2 Issues Report 3-96

DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS) 3-250

RAC/CAIS Version 1 Compliance Study 3-277

Ontologic Object Manager and KAPSE DBMS Requirements 3-349

An Object-Oriented DBMS for Design Support Applications 3-366

A Distributed CAIS 3-387

Extending the Granularity of Representation and Control for ~~the MIL-STD~~ CAIS 1.0 Node Model 3-398

NASA Space Station Software Requirements 3-409

An Interim Progress Report 3-422

~~ALS/N Multiprogramming Distribution~~ 3-463

Applying Denotational Semantics to Specifying Kernel Interfaces 3-498

Applying Semantic Description Techniques to the CAIS 3-509

Quality Assurance Guidelines 3-538

Ada Interoperability Survey 3-550

Draft KIT/KITIA Glossary 3-553

An Investigation of the Common APSE Interface Set (CAIS) on an IBM S/370 Running VM/CMS 3-568

An Investigation of the Common APSE Interface Set (CAIS) on an IBM S/370 Running VM/CMS (Briefing) 3-584

Implementation of a Prototype CAIS Environment 3-600

TRW CAIS Prototype — A Report to the KIT/KITIA 3-615

SBID As a Basis for CAIS Implementation 3-659

PCTE — A Basic for a Portable Common Tool Environment 3-667

Keywords: [illegible] Programming Support Environment, [illegible]

CONTENTS (continued)

PCTE — A Basis for a Portable Common Tool Environment Design Guidelines 3-704

Overview of PCTE — A Basis for a Portable Common Tool Environment PCTE Project Team 3-716

Ada-Europe — PCTE Conformance to the RAC 3-733

KITIA Executive Committee Meeting 3-764

Quality Assurance Guidelines 3-767

A DISTRIBUTED CAIS

Sue LeGrand

SofTech, Inc.

Abstract

The purpose of this report is to supply a framework for discussion of distribution issues for the Common APSE (Ada* Programming Support Environment) Interface Set (CAIS) design. A taxonomy of methods will be presented . Major distribution issues will then be identified and various resolutions discussed. The designers of the CAIS will decide which functions will actually be controlled by the CAIS model and which are to be implementation dependent.

Some assumptions must be made for the purpose of scoping this study. We assume that a distributed computing system is composed of computing resources called elements, and each element is composed of:

At least one processor capable of running a development task
At least one portable tool
A CAIS implementation.

The modular nature of ,the Ada language promotes dispersed development of large programs among many experts. A distributed CAIS provides for software development over geographically dispersed organizations and over the life cycle of the project and the development systems. Distribution enhances portability of code between heterogeneous systems and interoperability of data between them. That is, distribution of a development system may be the ultimate fulfillment of the CAIS model. Implementations will have a need for management of process control, scheduling, accountability, availability, fault tolerance and system configuration management and extendibility.

There is no limit to the amount of dispersion involved, and many applications require that the units be located all over the world and in outer space. Many military applications for command, control and communication have been defined. Twenty-two different kinds of networks have been identified so far for the NASA Space Station.

Private industry has begun to stress distributed computer systems as evidenced by the recent adoption of Manufacturing Automation

*Ada is a trademark of the U.S. Government, Ada Joint Program Office (AJPO).

Protocol (MAP) networks by many companies. [14] These networks use the standards defined by the International Standards Organization. [12]

The second version of CAIS, which is being produced now by SofTech, Inc., treats the issue of distribution. [7] Charles Howell at MITRE, Inc. has led a project to build a prototype of a distributed CAIS model. [8] Both of these efforts are producing reports which will be in the public domain.

Taxonomy of Distributed Systems

A taxonomy must consider many different aspects of distribution. Physical distribution is not defined in steps, but in a broad spectrum of system types that may or may not have various features and functions. Two systems may be comparably distributed and sophisticated but greatly different. Computer systems can be physically distributed in many different ways. The application, available resources and space and budget of the users will impact the decision. [3] The discussion below assigns classifications to the system configurations upon which a CAIS might be implemented.

Uniprocessor - Multiprogramming

This shall be designated Class A. Some instances of uniprocessors are often called distribution, but they are not distributed processing. Uniprocessors are mentioned here only for reference. Occasionally an office system is advertised which provides "distributed" data processing functions over an entire building. However, close examination shows the system to be merely many terminals all attached to one over-worked uniprocessor. In this system, a single processor is capable of multitasking and a number of users time-share. They access the central processor on a number of terminals located at various locations, but the terminals are all wired into the same computer. One processor is present and is responsible for all tasks: calculations, input/output functions, database management, etc.

Multiprocessors

This is class B. A number of processors are located together on the same board, or at least on boards in the same cabinet. This is distribution in the strictest sense, because the various processors cooperate, doing separate and parallel tasks, and they communicate through a medium. This system may be very big and complex and capable. All users still access this group of processors through the processor responsible for user interface.

Clusters

This is class C. A more obviously distributed system is organized around multiple homogeneous processing elements, or entities, that are physically separate but in close proximity. Each entity has at least one processor and associated software capable of performing automated, intelligent functions. These entities have different capabilities and will be organized on a functional and/or geographical basis, but they cooperate in the support of user requirements. The entities are connected through physical cables (usually high speed), and one processor in each entity is required to handle communication using the same protocol as the other entities in the cluster. The implication is dispersion of similar hardware and software. Data may still be in a central location (with an associated processor) and accessed through one data manager. An example of a cluster communication scheme is IBM's System Network Architecture (SNA).

Local Area Network (LAN)

This is class D. The entities are still connected by physical cable, but they may not have the same capability or even be homogeneous. The entities are usually autonomous, having control over their own domain only. They communicate through network interface units (NIUs) with separate hardware and software to translate the communication protocol to a standard protocol agreed upon and shared by the all of the entities on the entire LAN. There will be one net server for an entire cluster described above to share for net access. For instance, a cluster using SNA may be on the same net as a VAX cluster using DEC Net.

The NIUs will be responsible for translating each local protocol to one that the entire net can use. There are international standards for this which will be discussed later. Typical functions handled by the NIUs include modem, parallel to serial conversion, repeater, address recognition, parity check, token management, bit stuffing and error timeouts.

Data is distributed among the entities, and there are various schemes for users to access the distributed data base. This and the challenge of access control will also be discussed later. A user should be able to draw upon the entire collection of files in the distributed data base as though it were kept locally.

Remote Area Networks

This is class E. It is possible to tie clusters or even local area networks together and create remote area networks (RANs). These nets are connected by gateways that are responsible for any differences of protocol or hardware. There will be only one gateway to serve an entire cluster or LAN member of the RAN. A gateway often must provide high speed links over long distances with few occurrences of errors.

Advantages of centralized processing and distributed processing

A central computer source offers the greatest amount of control and the use of only one standard protocol for all users and all applications. There is no need or opportunity to duplicate software resources or data bases. There is no problem with incompatible hardware, since all components are extensions of the central hardware. Project control, programming techniques and configuration control are easily implemented. Vendor and other technical support is simple and less expensive than for systems with more than one computer.

Distributed processing has certain advantages over a central processor or even a central computing center. The most obvious is extendibility from a low cost initial investment. Small, specialized computers can be obtained and then added to as requirements and funds increase. These small systems are easier to use and maintain as well as less expensive.

Local processing can be done with much less delay than letting all users submit jobs through a central control. Results can then be sent to a larger unit for summarizing with other units' work. The result is also less data being transmitted. This is a form of parallel processing.

User control is often cited as a good reason for distributed processing. Every user has critical data that he likes to have located at his work station. Each user is also convinced that his choice of resources is the best. With distribution, there is a better chance of giving the user his selection of tools.

Distributed backup of computer resources gives the added protection against loss of an entire facility. Backup resources can be located far enough away to be safe. These backup resources can ordinarily be used to share the work load.

Space Station Requirements and Goals

The NASA Space Station requires a complex computer network of clusters, LANs and RANs. There will be a variety over time and location of system resources, system requirements and job requirements. There will be non-stop target systems performing non-stop functions. The hardware and software of these target systems will need to be upgraded, expanded and tested without bringing them to a halt. Process control will require elaborate interprocess communication and prioritizing. Load balancing and synchronizing of tasks will be sophisticated. Resources must be managed with strict access control. Some resources must be shared while others must be hidden and protected.

The Space Station Program goal of software maintenance is to be independent of the origination of the software and valid over the entire life cycle of the project and the host and target systems. The goal of user interface is to have a common interface for all user groups, resources and locations. The goal of accountability is to have a automatic record of all charges, credits and privileges common to the entire distributed network and transparent to the user.

CAIS Implications

The CAIS model is still an interface between layers, even when it is on a distributed computer system. The difference is that the underlying hardware is distributed and the tools include communication software. See Figure 1.

According to Richard Thall, Technical Director of the CAIS 2 development team, if the underlying hardware and software is a distributed system, then the CAIS will provide interfaces to allow development tools some limited control of the distributed environment. The services for distributed capabilities must be structured so that tools can be freely moved between monolithic and distributed systems. For example development tools imported from a monolithic system must not be forced to supply routing parameters to a distributed implementation; and tools from a distributed implementation must operate normally, when routing information is supplied to a monolithic system.

The goal of CAIS 2 is to allow tools to effectively operate in a distributed environment where the system elements may not be homogeneous. This means that the CAIS must supply:

a) a method of routing data to and from separately identifiable elements of a distributed system,

b) a method for exchanging and converting data having differing representations,

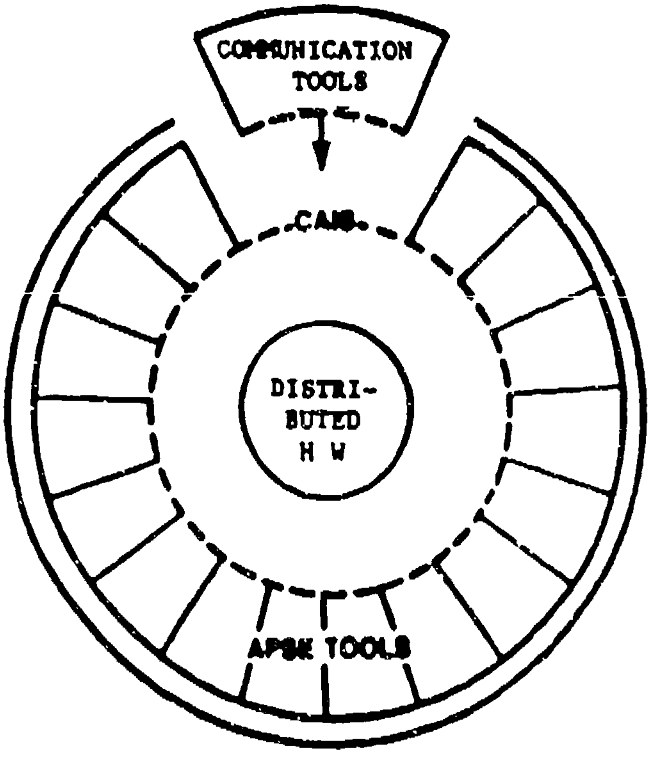


Figure 1. Distributed CAIS Model

c) methods for dealing with a database which may or may not be distributed in a transparent fashion, and

d) a method of initiating and controlling program execution on specific remote processors.

To preserve development tool portability, these services must be constrained to methods likely to be useful and useable in all distributed systems. Moreover, these services must have well defined behavior when implemented on monolithic systems.

This author believes that the best way that CAIS can provide these services is to use International Standards Organization (ISO) Open Systems Interconnection (OSI) defined communication tools. If the tools are built to ISO OSI standards and the CAIS interface to these tools is the same among implementations, then the communication tools will be portable. Suppose a development tool needed a software module that resided on a remote system. The Ada "with" command would list this module assuming it was in the local program library. The CAIS would call on the services of some communication tools to provide the modul. in a way transparent to the user.

The ISO OSI model

The ISO OSI model provides the international standards for any computer system to be open to communication with any other system obeying the same standards, anywhere in the world. It consists of a Reference model, Services and Protocols, each providing more constraining specifications but consistent with the others.

The Reference model defines types of objects in the system and general relations among these types. The Service Specifications define in more detail the service provided in each layer. The Protocol Specification is the lowest level of abstraction and defines precisely what control information is to be sent and what procedures are to be used to interpret this control information. It is the Protocol Specification that is of the most interest, since it defines the tools to be used for communication. See Figure 2. These tools will all need the support of the CAIS services, and the CAIS may depend on them to support the needs of other tools.

So far the ISO model includes tools specifically for each layer and for:

Naming and Addressing
Security
Commitment, Concurrency, Recovery
Job Transfer and Manipulation
Virtual Terminal, including Graphics
Fault Management
File Access and Management
Data Descriptive File

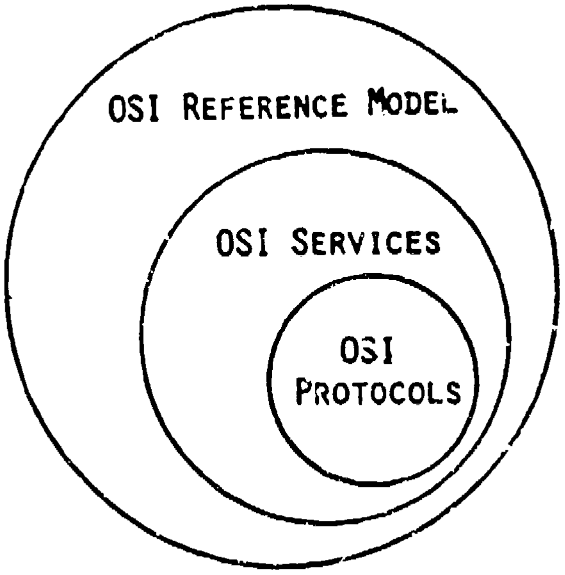


Figure 2. The ISO OSI Model

CAIS and Distribution

The issue of distribution is straightforward in the CAIS model. However, CAIS implementations must consider many aspects not shown in Figure 1. The issues of access, user transparency and fault tolerance must be considered. Communication methods impact the kinds of tools required and how they are used. Data Base Distribution can be accomplished many different ways, and this also affects the tools. Processes can be distributed over the physical network according to different rules. The interconnection control can be tight or loose coupling. The exchanges between various host and various target environments require different tools and different protocols. Each of these aspects affects the way the tools will interface with a system.

The following are discussions of possible communication implementation features that try to provide for the above needs. Each of these are implemented in tools that provide and require special services of the CAIS 2.

Agents

Agent tools are required to act on behalf of processes located on other systems. Suppose a process running on a system in New York needs the service of an array processor in Houston. There must be an agent on the New York system to work with an agent on the Houston system to coordinate the distributed processing. Agents are also needed for file transfer.

Network Integrity

Tools are needed to monitor and report network system health and status. They will consolidate reports and provide them to the network administrator at each location. They will need reports from the CAIS 2 services.

Accounting

There is often a requirement for reporting use of the network for accounting. Accounting tools must record which processes from which systems used the network and for what purposes and for how long. These tools will need reports from the CAIS services.

User Interface

Tools must exist to provide a friendly user interface and hide the mechanics of distribution. The CAIS model must provide the interface between these user features and the distributed operating system.

Access Control

Access control involves the usual security considerations. In a distributed environment it also involves considerations of the various capabilities of the systems and how they are implemented.[4] The ISO OSI provides procedures for access control.

Fault Tolerance

Tools that provide fault avoidance will need CAIS services to help with such things as error correction. Fault tolerance tools, on the other hand, will require services to help locate the backup element, bring the backups on line and synchronize the new participat on. The CAIS will provide these services or in turn rely on other tools.

Communication Connections

The type of connection between elements of the network will dictate the use of particular tools. In addition to the consideration of synchronous or asynchronous transmission, the system may use a number of different connections.

Point-to-point connections allow two and only two elements to talk directly with each other. Elaborate switching mechanisms are required with associated software tools. These tools will need the help of CAIS services to help identify the location of the elements, to assist the switching mechanism, and to sychronize the switching.

Store-and-forward connections depend on each element to receive all messages and send them on to the their destination. Each kind of mechanism to accomplish this must have an associated device driver. Tools are needed to aid message sending synchronization and to aid in identifying which messages to copy as they are passed and how to route new messages. The CAIS must see that these services are provided and coordinated.

Broadcast connections allow all elements to listen to all messages. Tools for this broadcast method will also require services provided by or obtained through the CAIS.

Message Handling

Modern networks send messages in packets. These packets are usually of uniform size and are preceded and followed by code that delimits them and assists in their handling. There are various ways to send these packets. A datagram is a message sent from the source to the destination one way, one time with no acknowledgement. It is good for very high speed connections that do not require a high degree of accuracy. Telephone messages are sent this way.

A virtual circuit appears to the user to be a point-to-point connection. Each packet is acknowledged from the receiver as it arrives. Some acknowledgements include code checking data.

The International Standards Organization (ISO) Open Systems Interconnection (OSI) model contains tools already defined for message handling. The CAIS must be the interface between these tools and the operating system and other tools.

Data Base Distribution

Traditionally data bases were considered in terms of shared files or distributed data. In a distributed system, both conditions exist at the same time. It is common to store a back up file copy at a remote node of the network under the control of another system with a different file structure. This copy will then need to be accessed and kept updated just as the original copy. File transfers must also be supported. Intertask allocation of shared files becomes more complicated in a distributed system. The tools to handle the data base management will depend on gate keeping provided by the CAIS.

Distributed Processing

Tools will be needed to start, stop, monitor and control processes distributed in other systems. Other tools will be needed to cooperate with remote systems upon which jobs are initiated that require local support of distributed processing. This is in addition to the remote access to resources such as files and array processors. Some tools for this feature are being defined by the ISO. Others will be needed and all will require CAIS services.

Interconnection

Tools will be necessary to manage the interconnection of the communicating systems. They may use tight or loose coupling. Tight coupling provides for a master and slave. It is often used when a number of intelligent terminals are tied to a central main frame. Each station may work independently, but all communication between them is tightly controlled by the main frame. Tools that facilitate this will need gate keeping services from the CAIS.

Loose coupling exists between completely autonomous systems. Communication occurs as a result of cooperation between the systems. The OSI provides for various states of this communication and provides the protocols and parameters needed. Tools involved in this type of interconnection will need the support of the CAIS services.

Virtual Terminal

In a distributed system there is often the requirement for a user to sit at a terminal in one location and access a system in another location as if his terminal was connected to it. A tool will be needed to work with the CAIS services in both systems to make this happen.

Host-Target Interface

Tools will be needed to provide the interface of hosts and targets. They will be needed for monitoring, debugging, and upgrading resources developed on the host for the target environment and the run time support for these resources.

The NATO Interface Set

The Nunn Amendment Special Working Group (SWG) on APSEs has a goal of defining a standard interface based on the CAIS model and using important contributions from the Portable Common Tool Environment (PCTE). This is an exciting concept, because development of the two models has concentrated on complimenting areas of concern. For instance, the CAIS model defines services of the KAPSE and is concerned with security. The PCTE model uses schemes and has begun treating the issues concerning distribution.

Representatives from both groups have been studying the similarities and differences. Other interested people have been tracking the progress of both groups hoping for contributions to the design of important projects. Dr. Charles McKay, of the University of Houston at Clear Lake recently sent constructive comments to both groups from the perspective of the NASA Space Station needs. Herm Fiscner, lead author of "A View of the CAIS from Industry and Academia", has worked closely with representatives from the group creating the PCTE.

If the NATO Interface Set incorporates facilities for a distributed CAIS and is accomplished in a timely manner, it may be one of the most important contributions to the Ada community.

BIBLIOGRAPHY

[1] Fisher, David & Richard Weatherly, "Issues in the Design of a Distributed Operating System for Ada", Computer, May 1986.

[2] Fischer, Herman, et.al., "Views on a CAIS from Industry & Academia", presented to the KIT, August 1985.

[3] Kleinrock, Leonard, "Distributed Systems", Communications of the ACM, November 1985.

[4] LeGrand, Sue, "An Access Control Model for a Distributed, CAIS-Conforming System", Proceedings of the AIAA/ACM/NASA/ IEEE Computers in Aerospace V Conference, October, 1985.

[5] LeGrand, Sue, "An Open Systems Interconnection Proposed for the Joint NASA/JSC UH-CL APSE Beta Test Site NOS Project and for the UH-CL NOS Test Bed Project Using the Upper Four Layers of the OSI Model", Master's Thesis, UHCL, May, 1984.

[6] LeGrand, Sue, "Communications Standards That Impact Aerospace Projects", presentation at NASA/JSC, April 1985.

[7] LeGrand, Sue, Richard Thall, "The CAIS 2 Project", Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, June 1986.

[8] MITRE Corp., "Rehosting and Distributing the CAIS", MITRE IR&D Project #96040, due November 1986.

[9] Mokhoff, Nicolas, "Local and Global Networks", Computer Design, September 1, 1985.

[10] McKay, C. W., "Some Thoughts for the First Step in Extending CAIS 1.4 to Meet the Needs of Applications Requiring Distributed Hosts and Distributed Targets", Submitted to the KIT/KITIA January, 1986.

[11] McKay, C.W., "Working Notes on PCTE, CAIS", Submitted to the KIT and others, July, 1986.

[12] Rauch-Hindin, Wendy, "Communication Standards: OSI is not a Paper Tiger", Systems & Software, December 1985.

[13] Rauch-Hindin, Wendy, "Flexible Automation - Factory Networking", Systems & Software, December 1985.

[14] Ryder, Michelle, Jerry Sagli & King Won, "Module puts factory devices on the MAP", Systems & Software, September 1985.

[15 Seider, Ross, "Token-bus networking links TOP with Shop", Systems & Software, July 1985.

Extending the Granularity of Representation and Control for the MIL-STD CAIS 1.0 Node Model

Kathy L. Rogers
Rockwell International
Space Station Systems Division

Introduction

The Common APSE (Ada[1] Program Support Environment) Interface Set (CAIS) [DoD85] node model provides an excellent baseline for interfaces in a single-host development environment (see Figure 1). To encompass the entire spectrum of computing, however, the CAIS model should be extended in four areas. It should provide the interface between the engineering workstation and the host system throughout the entire lifecycle of the system. It should provide a basis for communication and integration functions needed by distributed host environments. It should provide common interfaces for communication mechanisms to and among target processors. It should provide facilities for integration, validation, and verification of test beds extending to distributed systems on geographically separate processors with heterogeneous instruction set architectures (ISAs). This paper proposes additions to the PROCESS NODE model to extend the CAIS into these four areas.[2]

Rationale

The intent of the CAIS is to promote transportability and interoperability. The user interface should provide the same view of the system for a remote workstation connected through a network as for a directly connected terminal. Accessibility and finer granularity of the PROCESS NODE and QUEUE file information could provide processor performance measures during the design phase of the project, debugging information during the coding phase, and assessments of hardware and software changes during the

[1] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO).

[2] It is the intent of this paper to discuss some of the topics which were explicitly deferred in MIL-STD CAIS 1.0.

Extending the Granularity of Representation
and Control for the MIL-STD CAIS 1.0 Node Model

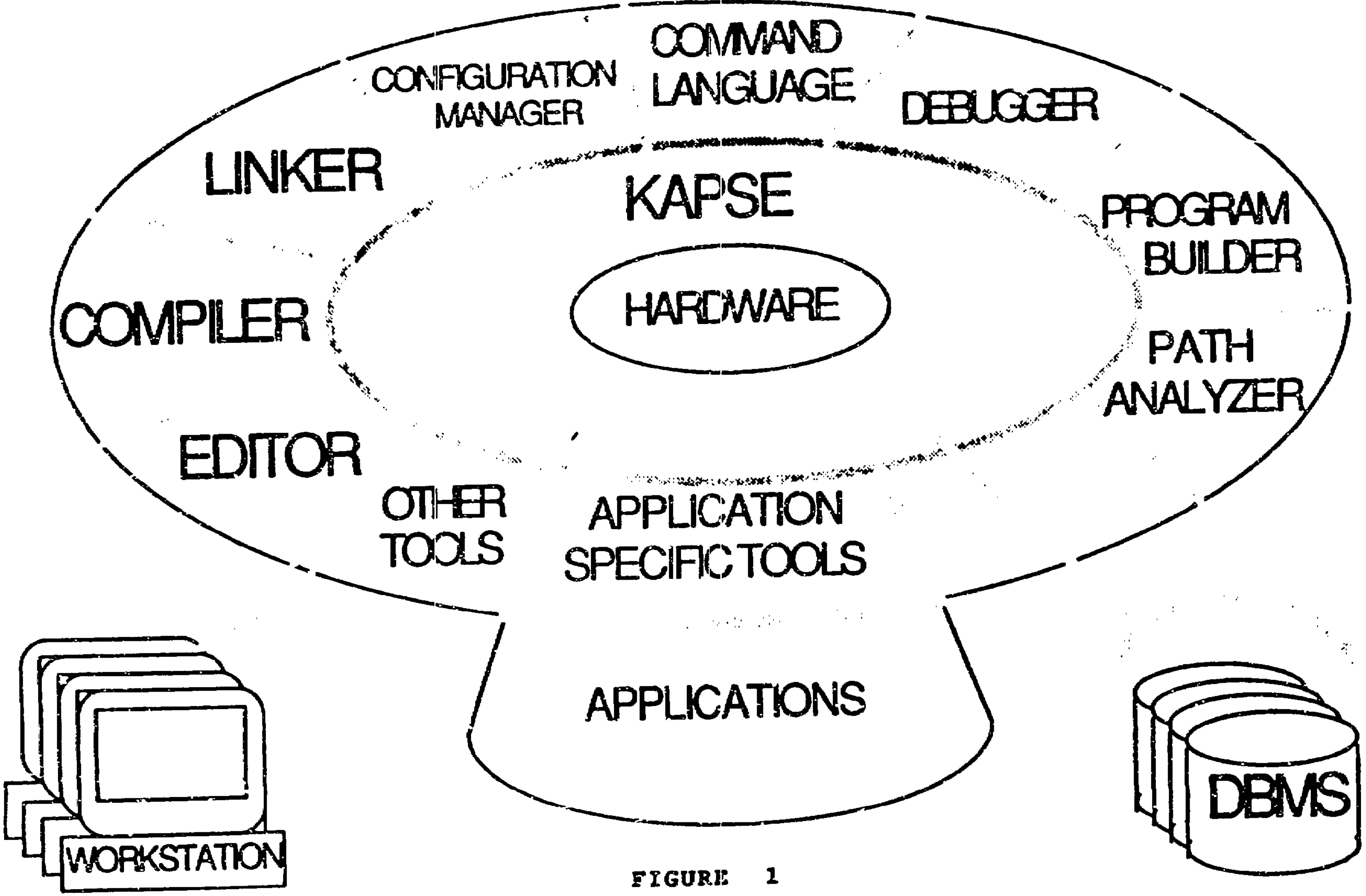


FIGURE 1

D.2.3.2

maintenance phase.

CAIS-provided code and data sharing could provide services (and entities) in a cost-effective manner to more than one application or user. To implement sharing, the node model must be able to manage data for dictionary driven processes, maintain version and revision information for library units, and provide security to maintain the integrity of the system. Data management requires information such as location, format, and access control of sharable resources. It might also extend to "knowledge" regarding the use of data, so that data may be relocated to facilitate convenient access. PROCESS NODES should be able to take advantage of code (such as common packages) that can be shared.[3]

The PROCESS NODE model should accommodate the communications necessary in a distributed environment. Five types of communication interfaces should be added to the current model: communication between parts of a process executing on separate processors, between processors (extending to processors with different ISAs), between the CAIS and the PROCESS (in both the host and the target environment), between different CAIS implementations, and among PROCESS NODES. In order to satisfy the Ada Language Reference Manual [LRM83] requirement that "several physical processors (may) implement a single logical processor"[4] effective information interchange is vital. Information must be communicated in an understandable format between heterogeneous ISAs. "Hooks" should be established so that individual elements of a test bed, as well as the integrated test bed, can be monitored. The CAIS should be extended to interact with other CAIS

[3] Multiple copies of packages, such as TEXT_IO, would be eliminated in favor of all processors at a site accessing the same copy. In a heterogeneous distributed environment, this can extend to shared copies of SYSTEM packages and STANDARD packages, if a common data representation scheme is used.

[4] Ada Language Reference Manual, Chapter 9, paragraph 5.

implementations.[5] When processes executing under the auspices of two different CAIS implementations interact and require CAIS services, a standard method should be used to determine which CAIS should be called. Interfacing to communication mechanisms, especially in a geographically separate system, is an important aspect of the CAIS.

Annotations for "non-functional"[6] directives could be handled by the PROCESS NODE model. These directives include desired degree of fault-tolerance, scheduling priority, desired level of status information, recovery processes, performance measures, special hardware requirements, and/or amount (and detail) of information to be promoted. Fault tolerance could be supported to ensure that sufficient resources are utilized to maintain the level of integrity required by the process. Scheduling of processes according to priorities should be considered; algorithms for serving processes according to their priorities could be provided in a straightforward manner. Directives stating the granularity of information required for a PROCESS (which determines the amount of overhead incurred) should be flexible.[7] Directives should also provide error recovery and rollback to the last "safe" state at a level of overhead which is appropriate for the PROCESS. Performance measures should be provided, especially for "time critical" processes which may need to be routed to a processor based on the speed and level of services available. The need to know the execution efficiency of processes on target processors is a major reason the CAIS services should be available in the target environment. In some configurations,

[5] Oberndorf, Patricia, Prototyping CAIS [Obern86].

[6] "Non-functional" is used here to denote constraints on functionality beyond those which are explicitly written into the code.

[7] For example, information pertaining to the current/last instruction or procedure executing might be requested. In the same way, the status of entities ranging from register values to values of user variables might also be requested.

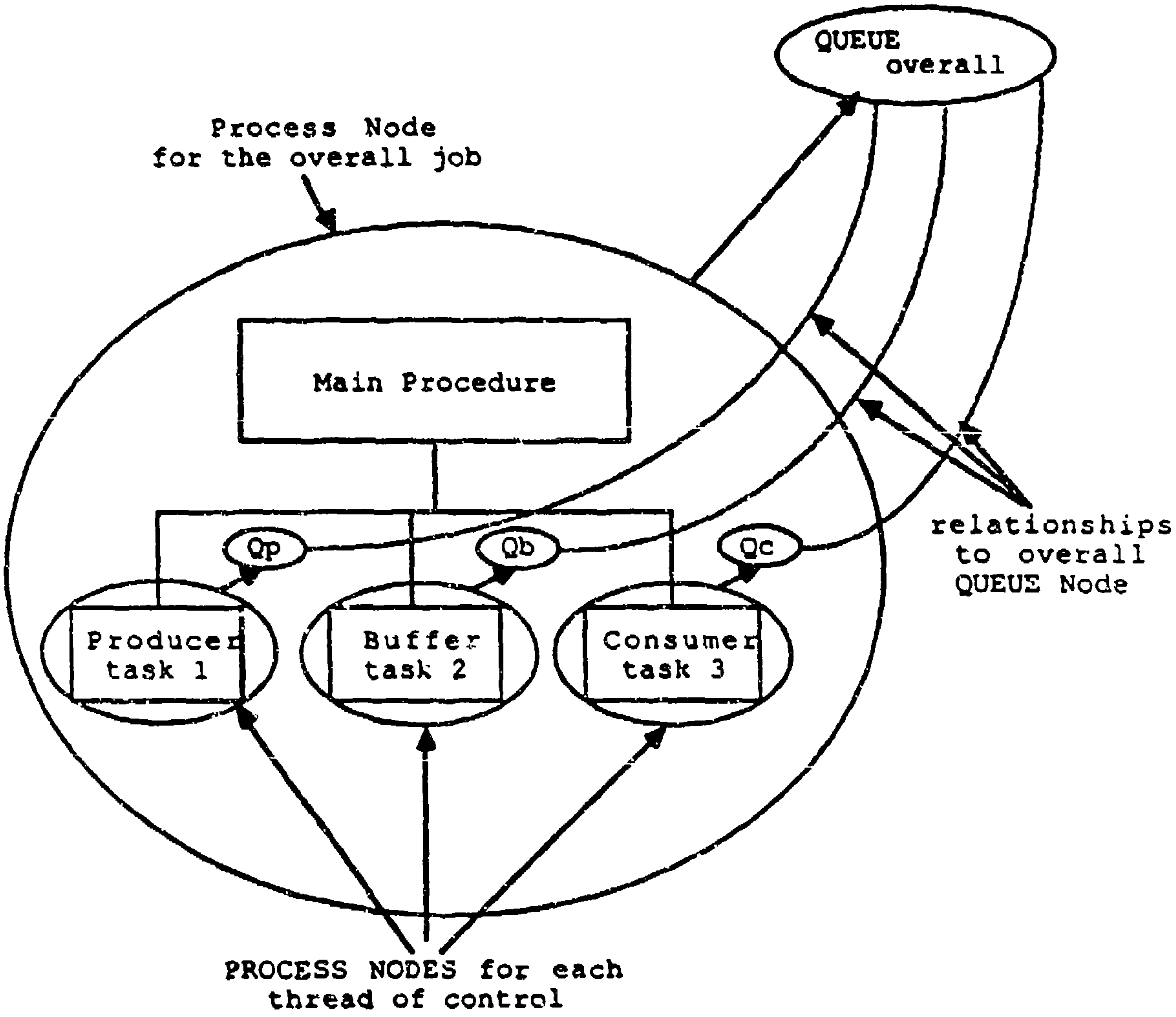


FIGURE 2
Snapshot of four PROCESS NODES

security will be important; security directives should be provided and enforceable [LeGra86]. The CAIS can be continually extended by providing additional handlers to accommodate future "non-functional" directives.

The PROCESS NODE model should include capabilities to query and to negotiate with other nodes. Negotiation may be required in the case of a remote procedure (subprogram) call where the size of the parameters exceeds the capabilities of the receiving processor. Query and negotiation procedures could detect this problem and establish a piecewise transmission of data. Processes executing on processors with different ISAs could negotiate a standard data format for transmitting data. Query capabilities are vital for processes which have very specific processor needs. Query and negotiation capabilities should be provided to determine the optimal processor configuration to execute a process. Library management, in a system containing heterogeneous ISAs and specialized processors, creates demand for information such as version/revision, intended ISA, special processing needs or priorities, and other required support.[8] Check out, with locking mechanisms, must be maintained for library units. Security for the items being managed is also a concern. The level of access required to read or update information must be established, including altering access requirements after updates. Creation and maintenance of multiple copies must be addressed with respect to update[9] procedures.

Recommendations

The current CAIS node model should be enhanced in four ways. First, the PROCESS NODE state information should be more descriptive. Second, there should be a PROCESS NODE representation of the status of each

[8] Other support may include speed, space, and/or security requirements, etc.

[9] Update is being used here to encompass all modification functions, addition, modification, deletion, etc.

Extending the Graunlarity of Representation and Control for the MIL STD CAIS 1.0 Node Model

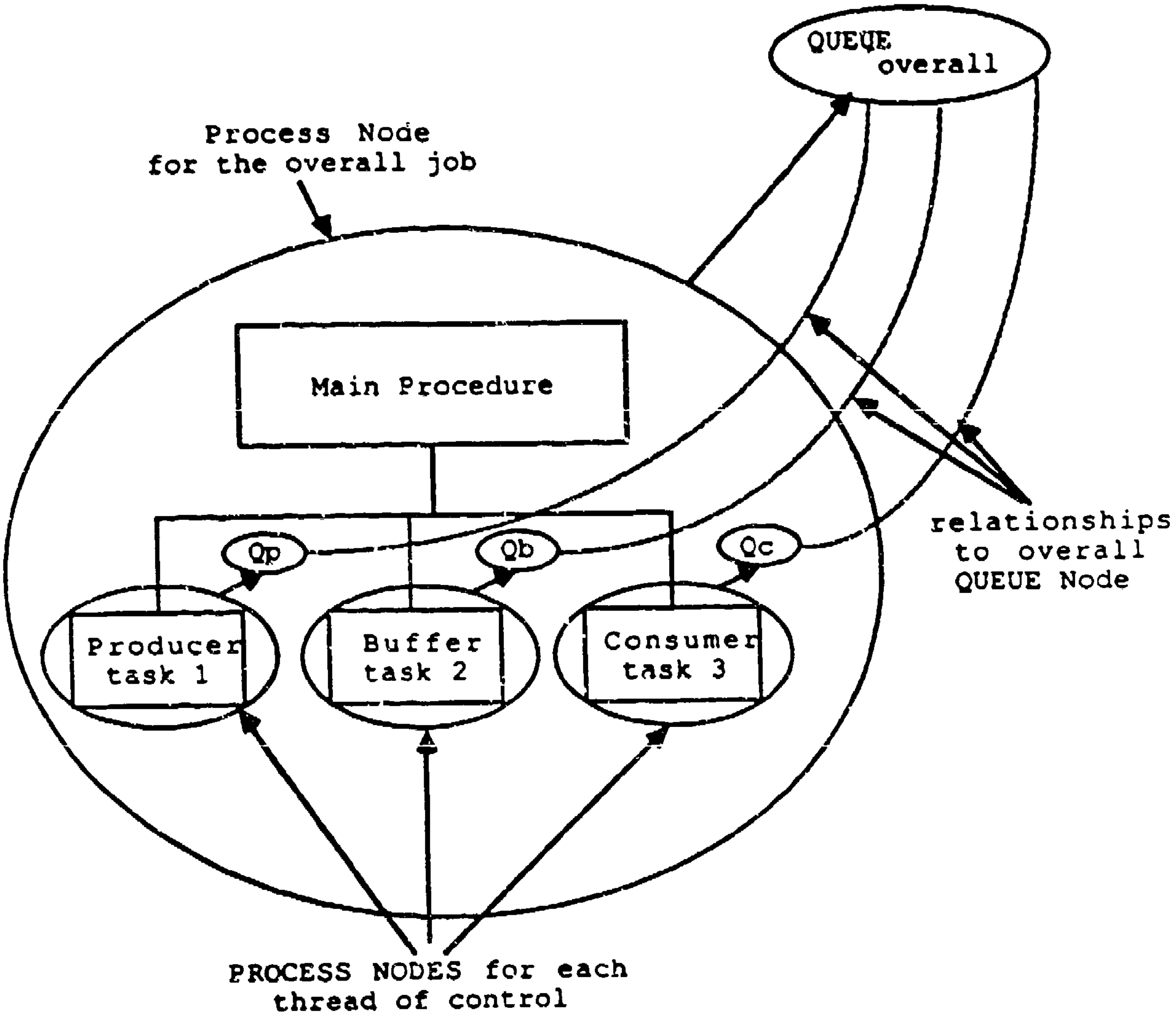

QUEUE overall
Process Node for the overall job
Main Procedure
Qp
Qb
Qc
Producer task 1
Buffer task 2
Consumer task 3
relationships to overall QUEUE Node
PROCESS NODES for each thread of control

FIGURE 2
Snapshot of four PROCESS NODES

thread of control extending to any level of decomposition, and a QUEUE associated with each PROCESS NODE. Third, the QUEUE NODE[10] should be able to provide accessible status measures beyond those which are "hard coded" into the process. Finally, the QUEUE NODE model should provide capabilities to act on the information received.

As an example of the implications of the above recommendations, consider a PROCESS that spawns three subordinate tasks: a producer, a buffer, and a consumer. Figure 2 is a snapshot of the four PROCESS NODES; it represents the state of each thread of control currently executing on behalf of the "main" process. The overall job, as well as each subordinate task is depicted as a PROCESS NODE, with an associated QUEUE NODE. Each PROCESS NODE has several predefined attributes including: CURRENT_STATUS, PARAMETERS, and RESULTS. Other information, such as the logical name of the site where the process is executing, may also be available. Each QUEUE NODE representing one of the subordinate tasks has a relationship to the QUEUE NODE associated with the PROCESS NODE for the overall job. Note that when the subordinate tasks terminate, their respective PROCESS and QUEUE NODES cease to exist.

In order to augment the PROCESS NODE, the process states should consist of "meta-states" as well as "micro-states". In addition to the current "meta-states" READY, SUSPENDED, ABORTED, and TERMINATED, a new meta-state, RUNNING, should be added. The meta-states should also have micro-states to provide additional information. The READY meta-state should include the micro-states WAITING (for resources), COMPLETE (but not terminated), and BLOCKED (awaiting rendezvous). The TERMINATED meta-state should include the micro-states NORMAL and ABNORMAL.

To increase the granularity of the PROCESS model, the PROCESS NODE, which represents the overall job should also provide PROCESS NODES for each "thread of control". That is, a PROCESS NODE should be associated with every body of a subprogram, task, or package in a state of execution. All PROCESS NODES should be of the

[10] The term QUEUE NODE is used (rather than QUEUE FILE) in order to describe the QUEUE as an entity.

same form (complete with proposed extensions).[11] The PROCESS NODE for each thread of control should have an associated QUEUE NODE. Information from QUEUE NODES should be promotable upward to the QUEUE NODE representing the next higher level of decomposition, based on the amount of information required by the higher level PROCESS/QUEUE pair. In this way, the current CAIS PROCESS NODE is maintained on the job level, but is also decomposable to provide more specific information when needed.

Status information provided to the QUEUE[12] should be usable by other processes. In the current model, data, procedures, or tasks in one process cannot be directly referenced from another process.[13] QUEUE files are currently used as holders of PROCESS information.[14] The level of detail for status messages and the amount of overhead incurred, should be able to be specified. Other specifiable information includes the amount of information that should be promoted from a QUEUE NODE at any level to a QUEUE NODE related to a PROCESS at a higher level. Extensibility of the QUEUE NODE model can be provided by viewing the node as a database which can be queried by applications (or engineers). Additional information could be added to the database in the future, which could be utilized by processes which are aware of the enhancements. Status information generated independent of the process (or processor) is necessary in a distributed system, in the event of process (or processor) failure.

The QUEUE should be more than a passive information receptacle. It should be capable of being used to initiate procedures, such as recovery upon

[11] The PROCESS NODES should extend to any level of decomposition necessary.

[12] CAIS Rationale and Criteria document.

[13] MIL-STD CAIS 1.0 p. 14.

[14] Three types of QUEUE files are defined. The QUEUE files can operate in SOLO (write append, destructive read), COPY (SOLO QUEUE with initial contents), and MIMIC (dependent upon another QUEUE file) modes.

used to initiate procedures, such as recovery upon detection of a fault in the system. Facilities such as those necessary to terminate processes which are not performing correctly could also be provided. Early warning regarding process failure (rather than fault detection upon request for service) provides the calling process with a potentially greater number of recovery possibilities. Action in the event of failing processes is essential in environments which require fault tolerance, especially in unattended systems or in those systems where life and property depend on continuous, correct functioning of hardware and software.

Conclusion

The potentially long lifetime and large number of host development environments and target processor configurations, using Ada, require a CAIS that promotes transportability, interoperability, communication, and extensibility. The CAIS should provide a constant view (at an appropriate level of detail) of the supporting hardware and the APSE tools. This view should be provided to an engineer at a workstation, as well as to a secure, fault-tolerant distributed process. The CAIS should be extended to provide query and negotiation capabilities among nodes. It should include mechanisms for handling "non-functional" directives (in order to address the spectrum of processing complexity). It should also accommodate sharing code and data, as well as communication interfaces. These enhancements are necessary to accommodate the potential changes that will occur throughout the lifecycle of Ada applications. Some extensions to the CAIS model are necessary. Recommendations include maintaining more descriptive PROCESS state information; viewing the current PROCESS NODE model as a description at the overall job level (and providing PROCESS nodes for subprograms, tasks, and packages, while they possess a thread of control); viewing the QUEUE as a resource NODE rather than a logging file, and enhancing the QUEUE NODE to make it responsive to processing requirements. The proposed extensions to the CAIS model maintain the job level view of the original CAIS design and enhance it by providing decomposition to a finer level of granularity.

Extending the Granularity of Representation and Control for the MIL-STD CAIS 1.0 Node Model

The author gratefully acknowledges the contributions of the Joint NASA-JSC/UH-CL APSE Beta Test Site Team members, especially the expertise and encouragement of the research team's technical director, Dr. Charles W. McKay.

NASA

SPACE STATION

SOFTWARE REQUIREMENTS

EDWARD CHEVERS

JOHNSON SPACE CENTER

JANUARY 1986

INFORMATION IN THIS PAPER IS BASED IN PART ON WORK PERFORMED

BY

CHARLES McKAY AND RODNEY BOWN

UNIVERSITY OF HOUSTON- CLEAR LAKE

AND

PRESENTED IN A PAPER TITLED

ADA RUN TIME SUPPORT ENVIRONMENT

AND A

COMMON APSE INTERFACE SET

AT THE

AIAA COMPUTERS IN AEROSPACE CONFERENCE

OCTOBER, 1985

ADA BACKGROUND AT NASA

- MEMORANDUM OF AGREEMENT BETWEEN NASA (OAST/JACK KERREBROCK) AND DoD (EDITH MARTIN) FOR COOPERATION UNDER STARS PROGRAM

- EVALUATION OF DoD ADA LANGUAGES FOR FUTURE NASA PROJECTS FIRST TASK UNDER NASA/DoD STARS AGREEMENT

- TASK ASSIGNED TO JSC AND UH/CL HIGH TECHNOLOGY LABS PUT UNDER CONTRACT FOR BETA SITE SUPPORT

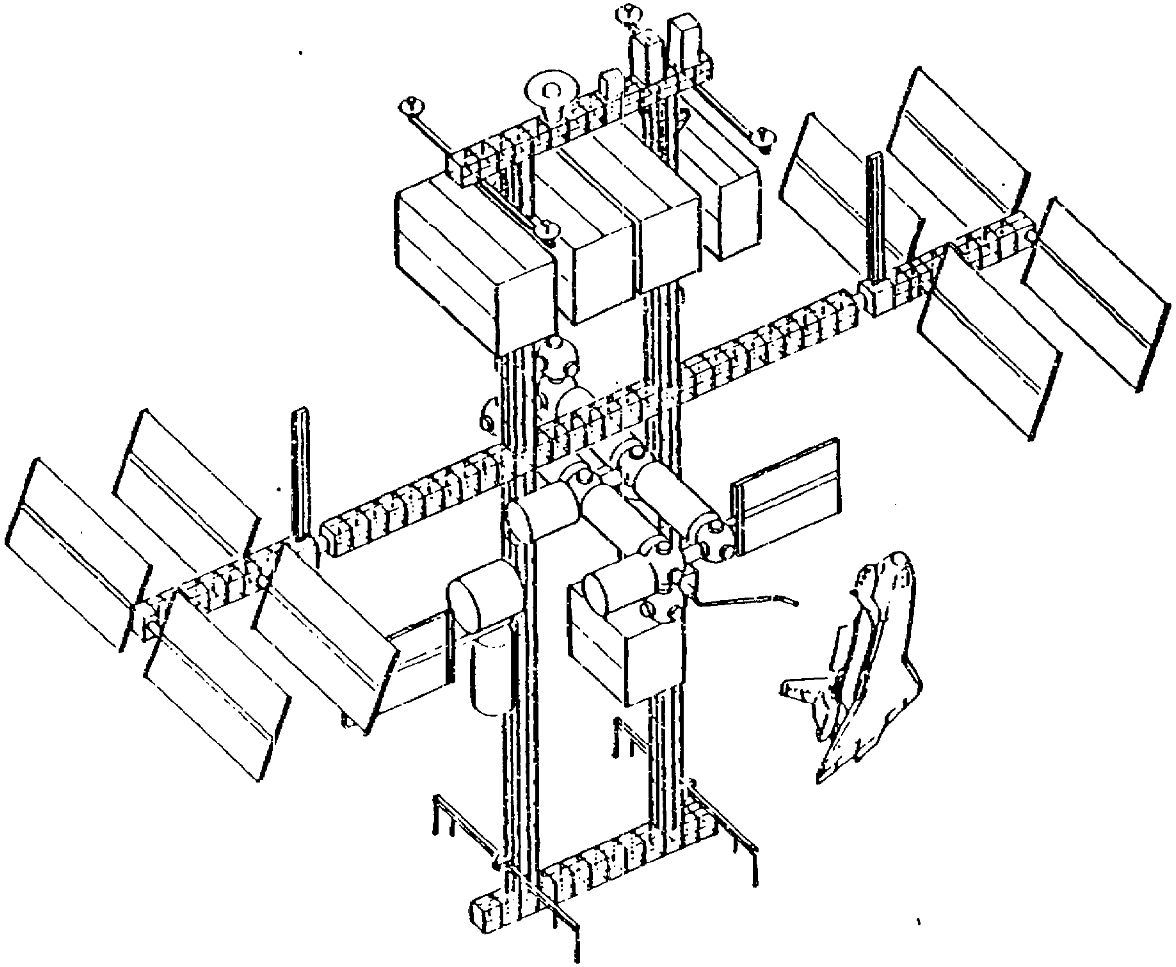
- SPACE STATION DMS TEST BED IDENTIFIED AS TARGET OF STUDIES AND PROJECT OPENED TO LOCAL JSC AEROSPACE COMMUNITY THRU UH/CL CONTRACT

- LANGUAGE EVALUATION EXPANDED TO INCLUDE COMMERCIAL COMPILERS IN ADDITION TO DoD ADA ENVIRONMENTS

ADA STATUS AT NASA

- JSC FUNDED INITIAL INSTALLATION OF ADA SYSTEMS AT GSFC, KSC, MSFC, JPL, LaRC, NSTL AND LeRC
- 65 ADA TASKS (ATOP's) DEFINED BY 25 AEROSPACE COMPANIES, UH/CL AND JSC FOR HOUSTON BETA SITE PLUS 3 TASKS FROM ESA (EUROPE)
- 10 ADA SOFTWARE SYSTEMS OPERATIONAL ON JSC/UH-CL NETWORK
- BASED ON ABOVE EFFORTS PLUS SPECIFICALLY DIRECTED STUDIES BY SPACE STATION PHASE B CONTRACTORS, JSC LEVEL C PROJECT OFFICE PROPOSED ADA AS BASELINE LANGUAGE ON JULY 19
- LEVEL B AGREED WITH THE RECOMMENDATION AND MADE ADA THE LANGUAGE OF CHOICE FOR ALL FLIGHT APPLICATION SOFTWARE ON AUGUST 15 (EXCLUDED OPERATING SYSTEM, DBMS, USER INTERFACE AND GROUND CONTROL/DISTRIBUTION)

Full Operational Capability Requires Integration of Payloads and Servicing Facilities



FULI OPERATIONAL CAPABILITY

SPACE STATION SOFTWARE SCHEDULE

- SPACE STATION MAJOR MILESTONES
  - PHASE B DESIGN DEFINITION STUDIES ... APR 85 TO JAN 87
  - PHASE C DETAIL DESIGN/FABRICATION ... START MID 87
  - FIRST DEPLOYABLE TRUSS LAUNCH ... LATE 1994
  - INITIAL OPERATIONAL CAPABILITY ... LATE 1996

- SPECIFICATION FOR A SOFTWARE SUPPORT ENVIRONMENT (SSE) WILL BE ISSUED IN APRIL 86 WITH CONTRACT AWARD IN NOV/DEC 86
  - * THIS INCLUDES THE COMPILER AND TOOLS REQUIRED BY PHASE C CONTRACTORS TO DEVELOP SUBSYSTEM APPLICATIONS CODE
  - * SSE ALSO USED BY PHASE C DMS CONTRACTOR FOR STANDARD DATA PROCESSOR SYSTEM SOFTWARE AND NETWORK OPERATING SYSTEM

- INITIAL SSE MUST BE DEFINED AT THE BEGINNING OF PHASE C (7/87)

- PROTOTYPE SSE SOFTWARE AND HOST FACILITY REQUIRED BY PDR (1/88)

- OPERATIONAL SSE/HOST WITH FULL CONFIGURATION CONTROL NEEDED BY SDR (7/88). THIS INCLUDES CODE GENERATORS AND SYSTEM VERIFICATION TECHNIQUES FOR DISTRIBUTED, HETEROGENEOUS PROCESSORS

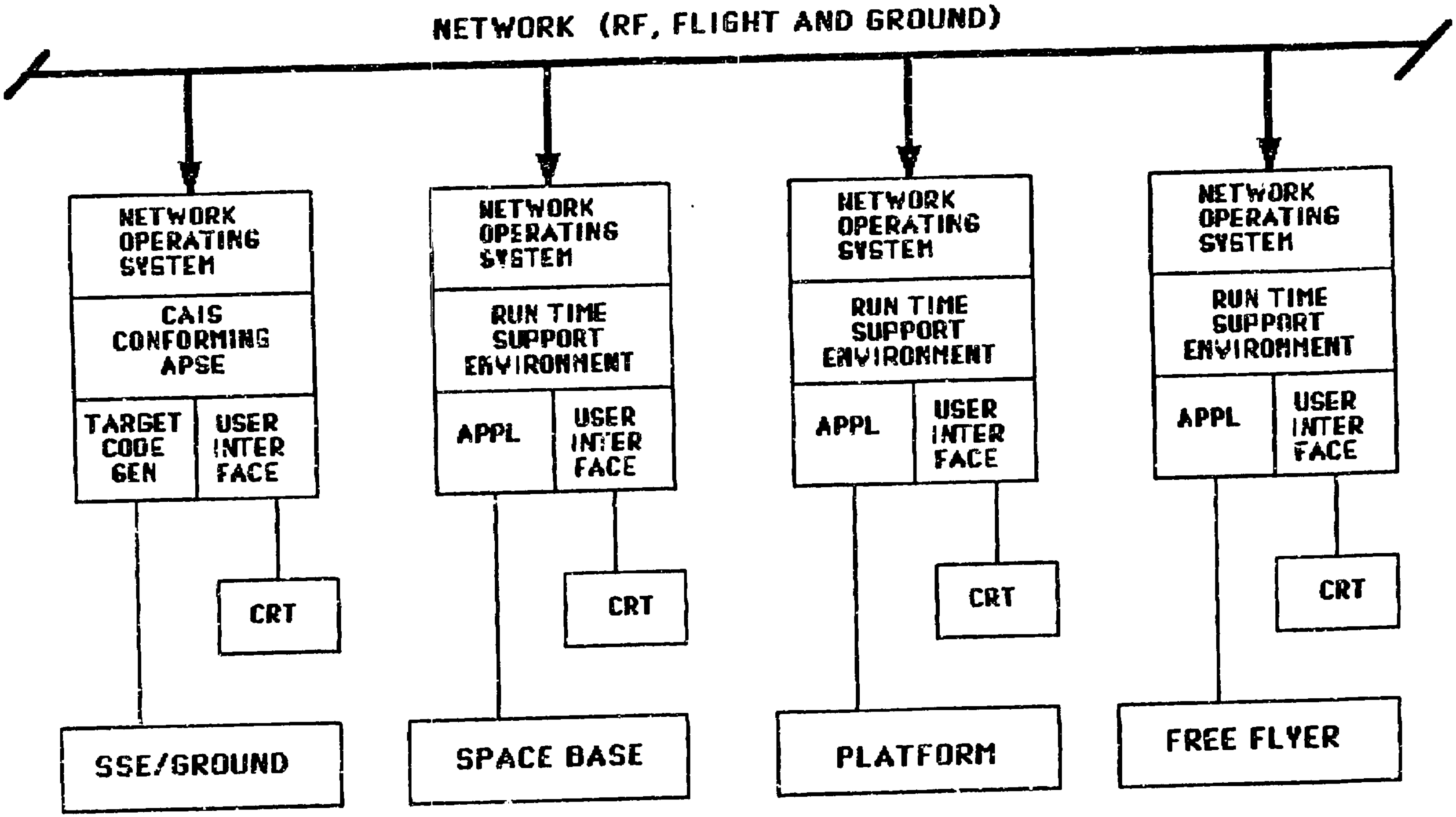
IMPLICATIONS OF SPACE STATION REQUIREMENTS

- SSE IS JUST ANOTHER NODE IN SS INFORMATION MANAGEMENT NETWORK

- SSE OPERATING SYSTEM IS PRIMARILY SOFTWARE DEVELOPMENT ORIENTED WHEREAS OTHER NODES ARE AKIN TO REAL TIME PROCESS CONTROLLERS

- THUS IT APPEARS THAT THE SSE HOST SHOULD SUPPORT THE "VIRTUAL ADA MACHINE" CONCEPT PROPOSED BY McKAY AND BOWN IN THEIR AIAA AEROSPACE COMPUTER CONFERENCE AT LONG BEACH
  - * IMPLICATIONS OF THIS CONCEPT ARE THAT THE ADA RUN TIME SUPPORT ENVIRONMENT (ARTSE) AND COMMON APSE INTERFACE SET (CAIS) ARE NOT INDEPENDENT FUNCTIONS
  - * REQUIRES HOST ENVIRONMENT TO BE CAIS CONFORMING AND A SINGLE WORK STATION FOR DEVELOPMENT OF BOTH THE RTSE AND APPLICATION CODE FOR DESIGNATED TARGET MACHINES
  - * THIS WILL REQUIRE ADDITIONS TO THE RUN TIME LIBRARY (PROBABLY IN AN EXTENDED RTL) PLUS CAPABILITY TO DEFINE CHARACTERISTICS OF A DISTRIBUTED SET OF TARGET PROCESSORS WITHIN THE SPACE STATION INFORMATION NETWORK

SPACE STATION INFORMATION NETWORK



NETWORK (RF, FLIGHT AND GROUND)

NETWORK OPERATING SYSTEM

CAIS CONFORMING APSE

TARGET CODE GEN

USER INTER FACE

CRT

SSE/GROUND

NETWORK OPERATING SYSTEM

RUN TIME SUPPORT ENVIRONMENT

APPL

USER INTER FACE

CRT

SPACE BASE

NETWORK OPERATING SYSTEM

RUN TIME SUPPORT ENVIRONMENT

APPL

USER INTER FACE

CRT

PLATFORM

NETWORK OPERATING SYSTEM

RUN TIME SUPPORT ENVIRONMENT

APPL

USER INTER FACE

CRT

FREE FLYER

SPACE STATION REQUIREMENTS ON SOFTWARE

- SPACE STATION CONSISTS OF DISTRIBUTED SET OF ENTITIES IN SPACE AND ON THE GROUND

- SOME EXPERIMENTS WILL REQUIRE COORDINATED DATA SETS FROM FREE FLYING MODULES, SPACE BASE, AND GROUND LOCATIONS

- COMPUTATIONS MAY NOT BE SPLIT IN THE INITIAL OPERATIONAL PHASE BUT DATA ACQUISITION WILL REQUIRE COOPERATION FROM ABOVE ELEMENTS

- DURING EARLY BUILD-UP, RENDEZVOUS AND DOCKING WILL BE A COORDINATED TASK BETWEEN GROUND CONTROL, THE SPACE BASE, AND THE SHUTTLE. AS THE STATION ACHIEVES OPERATIONAL STATUS THE GROUND CONTROL FUNCTION WILL GRADUALLY MOVE ONBOARD AND THE STATION MAY EVEN ASSUME CONTROL OVER THE ORBITER DURING FINAL APPROACH AND DOCKING

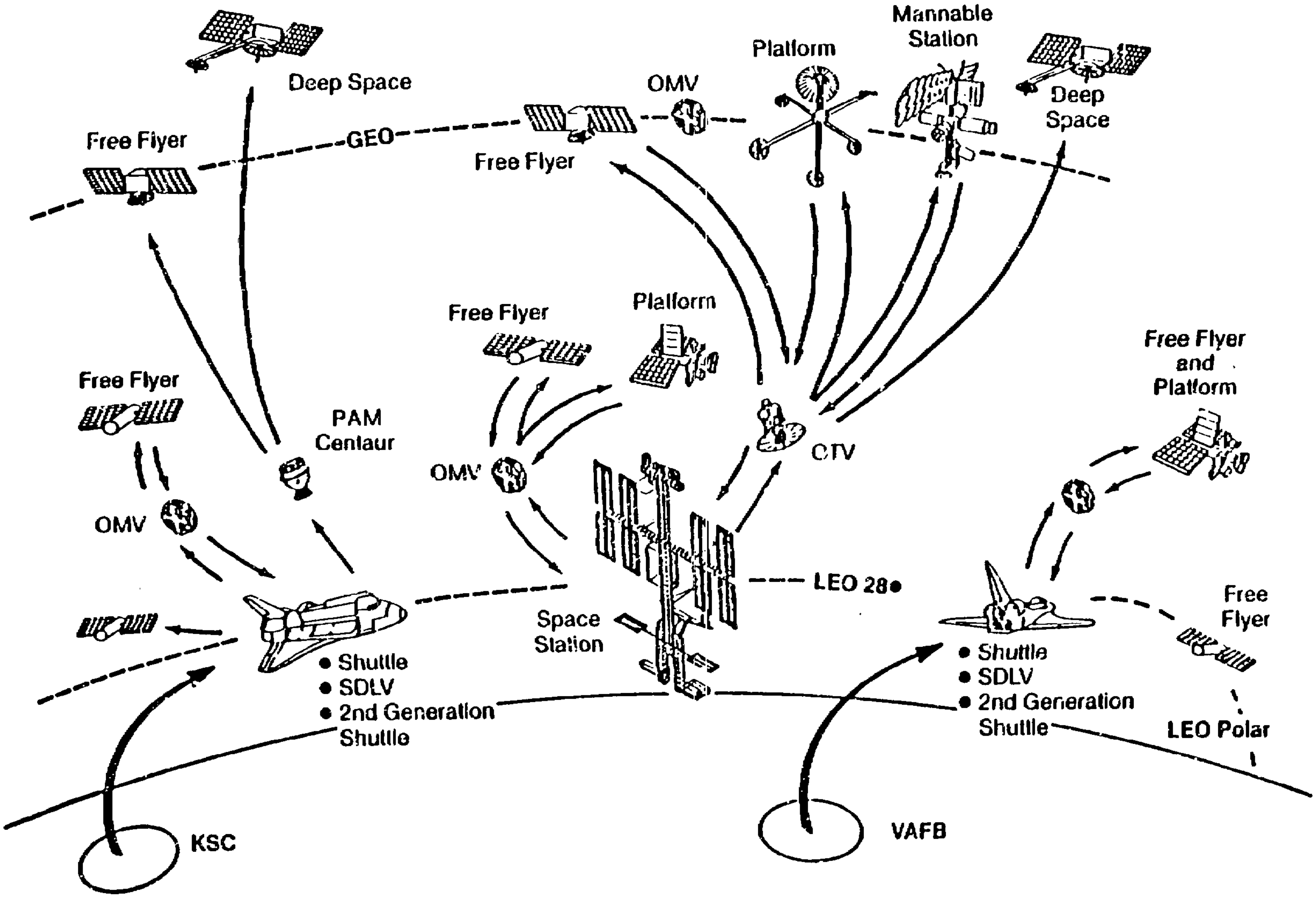


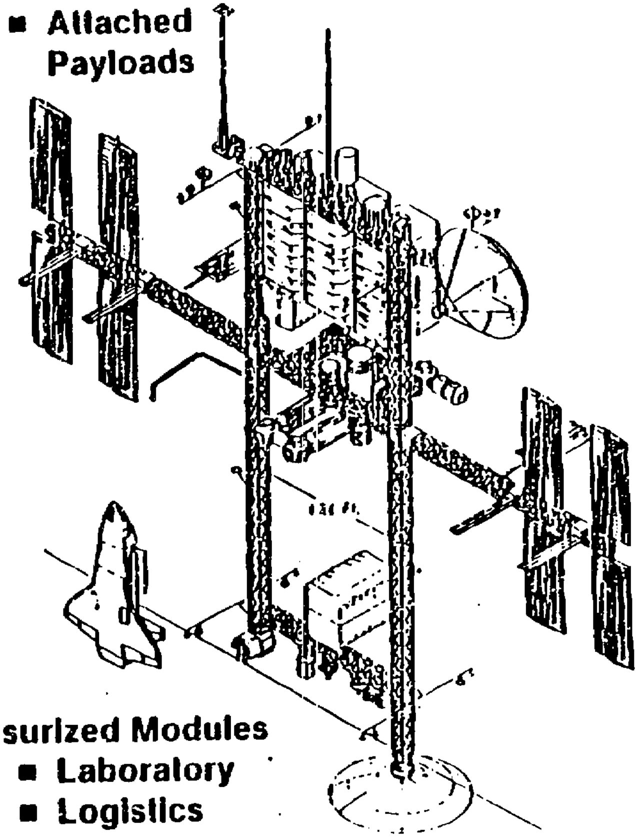
Figure 3.2-1. Space Station Will Evolve to a Permanent Headquarters in Space

VHP248

SPACE STATION PROGRAM SERVICING REQUIREMENTS

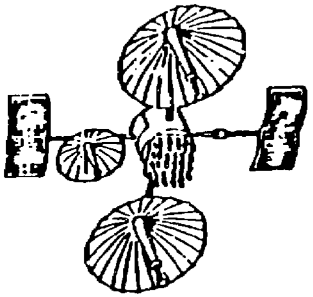
Space Station

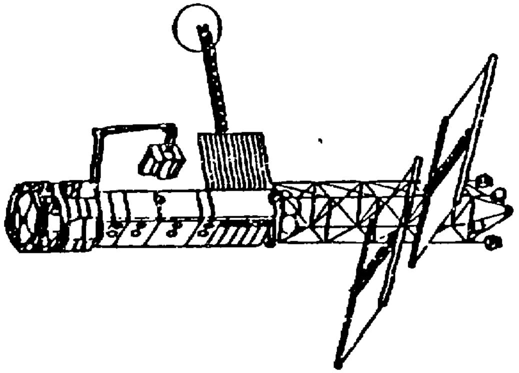
- Attached Payloads
- Solar Arrays



Satellites

- LEO
- GEO (Far Term)





Platforms

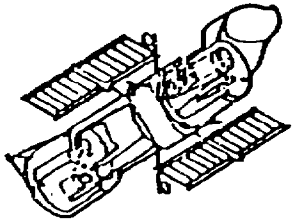
- Co-Orbiting
- Transportation Vehicles
  OMV
  OTV

Pressurized Modules

- Laboratory
- Logistics
- Habitation
- Commercial (EUS)

Mission Satellites

- Space Telescope
- AXAF
- LDR
- etc.



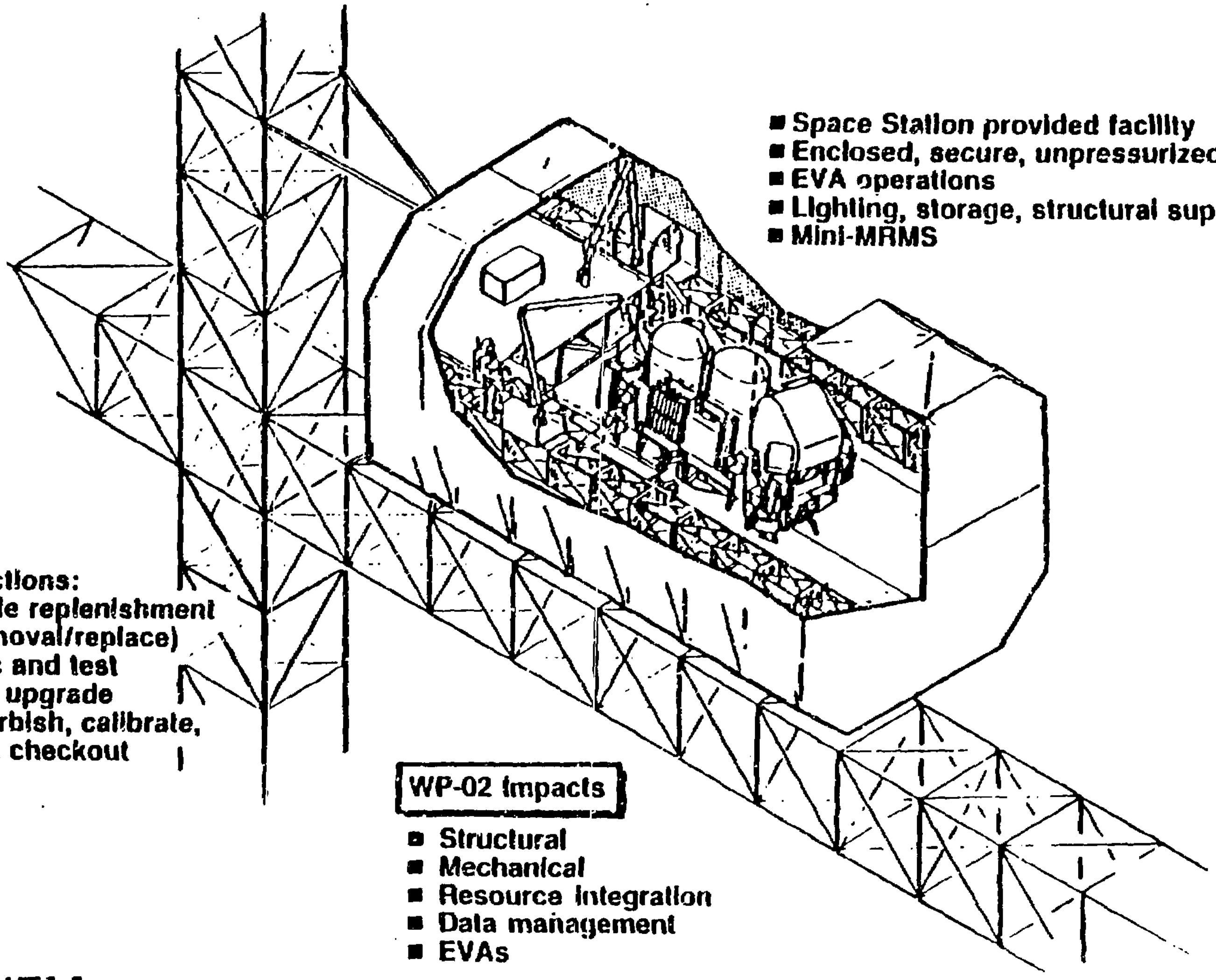
MCDONNELL DOUGLAS | IBM | Honeywell | RCA | Lockheed — SPACE STATION PROGRAM

MCDONNELL DOUGLAS CORPORATION

Space Station Customer Accommodations

VHG986

PAYLOAD SERVICING HANGER (GENERIC DESCRIPTION — PAST NASA CONTRACT)



- Space Station provided facility
- Enclosed, secure, unpressurized site
- EVA operations
- Lighting, storage, structural support
- Mini-MRMS

Service Functions:
- Consumable replenishment
- Repair (removal/replace)
- Diagnositic and test
- Equipment upgrade
- Clean, refurbish, calibrate, adjust, and checkout

WP-02 Impacts
- Structural
- Mechanical
- Resource integration
- Data management
- EVAs

IMPLICATIONS ON KIT/KITIA AND ARTWG

- THE SPACE STATION REPRESENTS A MAJOR INVESTMENT BY THE U.S. GOVERNMENT IN ADVANCED SOFTWARE PROGRAMS BASED ON THE ADA LANGUAGE

- NASA DOES NOT BELIEVE DEFERRAL OF DISTRIBUTED ENVIRONMENTS IS IN THE BEST INTEREST OF THE ADA LANGUAGE WHEN THE STATED GOALS OF ADA ARE TO SUPPORT DISTRIBUTED, EMBEDDED PROCESSING SYSTEMS

- IN THIS CONTEXT SOME OF THE DECISIONS AND POLICIES BEING MADE BY KIT/KITIA, ARTWG, WG-9 AND OTHER TEAMS/WORKING GROUPS HAVE SIGNIFICANT IMPACT ON BOTH THE SPACE STATION AND DoD PROGRAMS

RELATIVE TO THE SPACE STATION

* TO ADEQUATELY SUPPORT DISTRIBUTED PROCESSING INTER-TOOL INTERFACES, i.e. INTERACTION BETWEEN RUN TIME SYSTEMS AND DEBUGGER TOOLS SHOULD ALSO BE ADDRESSED AND NOT DEFERRED
* FINALLY, THE INTEROPERABILITY VIA DEFINITION OF EXTERNAL REPRESENTATION OF DATA FOR TRANSFER BETWEEN ENVIRONMENTS OR A HOST AND TARGET MUST ALSO BE CONSIDERED IN EARLY CAIS VERSIONS

An Interim Progress Report
on

"A Study to Identify Tools Needed to Extend the Minimal Toolset of the Ada* Programming Support Environment (MAPSE) to Support the Life Cycle of Large, Complex, Distributed Systems Such As the Space Station Program"

Task Order No. UHJSC B11

Contract No. NAS 9-17010

Team Leader: Charles McKay,
UHCL High Technologies Laboratory

Team Members: Robert Charette, SofTech
David Auty, SofTech

* Ada is a registered trademark of the Ada Joint Program Office and U.S. Government

2.0 Relevant Issues and Assumptions

It is the professional opinion of the team leader that each of the following issues and assumptions offers the potential to:

- lower the life cycle costs of the SSE and the SSP software
- lower the life cycle risks associated with this software
- improve the quality of this software and the productivity of those who produce and maintain it
- improve the state-of-the-practice of free world software technology and thus increase the free world's return-on-investment in the Space Station Program.

This opinion is that of Dr. McKay and should not be interpreted as reflecting positions by the University or SofTech or by Dr. McKay's teammates at SofTech or the University.

2.1 The Ada programming language will serve as the cornerstone for developing software for the SSP.

Elaboration:

Of all the languages which are currently available as either an ANSI or an ISO standard, Ada is:

- technically superior in the support of modern software engineering principles
- economically and politically superior in its potential to leverage free world advancements in software engineering during the development cycle of the SSP.

2.2 The MAPSE is the most appropriate beginning point for extending a minimal toolset for a development environment into a highly automated SSE for the life cycle of the SSP.

Elaboration:

The KAPSE (Kernel for the APSE) provides an interface to a virtual Ada host computer with an underlying project object base. Above the KAPSE, six host-independent functions are provided :

a compiler, an editor, a command language, a linking loader, a debugging tool suite and a configuration management capability.

2.3 Models which:

- o complement the Ada culture (ie, the Ada language and the software engineering goals and principles upon which it is are based)
- o are technically superior to currently competing models
- o are politically and economically viable to leverage over both the near and long terms, should be adopted and exploited. These include the following:
  - a) the Entity-Attribute/Relationship-Attribute (EA/RA) model
  - b) the Integrated Model for programming support environments
  - c) the Open Systems Interconnection (OSI) model for interconnecting distributed hosts and targets within the integrated software support environment

Elaboration:

The comparatively rapid growth in the use of the EA/RA (also called "ERA") model as reported on several successful projects is due to both its simplicity and its power. Its elegance is also visible in its proven ability to subsume and support the earlier data base models: hierarchical, network and relational.

The Integrated Model for programming support environments addresses large, complex systems by dividing life cycle concerns into three interrelated sets (processes, products and professional) and then by addressing them from two life cycle perspectives (activities and phases). (Definitions and further elaboration are given in section 3.0 of this report.) The Integrated Model then allows complementary methodologies and tools to be assembled to provide complete and consistent support of software engineering throughout the project's life cycle. The method of integration is achieved via a persistent model of the "life cycle project object base". By contrast, the Open Model for programming support environments (PSE's) allows the assembly of off-the-shelf components, tools, and tools sets from a variety of sources without requiring the collection to provide or support:

- a model of the life cycle
- complementary methodologies and tools
- complete and consistent support of Software Engineering throughout the life cycle
- integration via a persistent model of the life cycle project object base

The Open Systems Interconnection model (not to be confused with the Open system model for PSE's described above) provides a network of virtual services to be superimposed over a distributed collection of heterogeneous computing resources. These services can (and should) be part of an integrated model of an SSE.

2.4 Standards and draft standards which :

- complement the Ada culture
- are technically superior to currently competing standards

o are politically and economically viable to leverage over both the near and long terms should be adopted and exploited. These include the following:

a) ANSI/MIL STD 1815A (the Ada language)

b) DIANA (draft proposal) (Descriptive Intermediate Attributed Notation for Ada)

c) IRDS (draft proposal) (Information Resources Dictionary System)

d) CAIS (draft proposal) (Common APSE Interface Set)

e) ISO Standards for the OSI Model (standards for lower levels, draft standards for upper levels)

f) DoD STD 2167 (approved standard for life cycle documentation)

g) DoD STD 2168 (draft standards for life cycle quality assurance)

Elaboration:

In several other reports to NASA, the team leader has recommended (and explained the rationale for) items a), b), d) and e). Therefore this elaboration will be confined to c) IRDS, f) 2167, and g) 2168.

Every large, complex system is likely to be subdivided into several major subsystems and components which evolve over time. Typically these major subsystems and components benefit from a schema (defining structure) and a dictionary (defining elements). Both defining entities may-be/should-be described via the EA/RA model. (See assumption 2.3.)

There are obvious advantages to a standard, self-describing structure (which is extensible by a set of rules prescribed within the standard) for both the schema and the dictionary. (For example, such structures would facilitate information exchanges within upward compatible system evolutions and among heterogeneous systems.) The IRDS supports the application of the EA/RA model to such structures.

A major portion of the expense and the errors in the life cycle of today's systems can be attributed to documentation. DoD-STD-2167 defines a life cycle model and the documentation required for the associated phases and activities. It also defines the rules for tailoring and/or extending the standard to apply it to a particular application. In short, it is intended to:

- o reduce the quantity of required documentation
- o increase both the quality and the useability of the documentation that is required
- o increase the automated support of both technical and management functions within an integrated environment

Properly viewed, this is a subset of the broader issue of reuseable components and is a major factor in advancing the benefits of Software Engineering.

Draft Dod-STD-2168 is closely related to the already approved (within DoD) 2167. Since true quality assurance is possible only when there are standards, guidelines and practices to measure against, and since the observable (ie, quantifiably measurable) outputs of each life cycle phase are defined in its 2167-required documentation, this is a natural complement to the documentation standards.

Because of the well-defined standards (which are not subject to requests for waivers on future projects), automated tool support will be added to the MAPSE to facilitate the generation and continued adaptation of documentation generation and quality management a la' 2167 and 2168.

2.5 Both the object code produced by the SSE and the interactive command language between the host and the target environments should be based upon a catalog of standard interface features and options for the runtime support environment of a virtual Ada machine.

Elaboration:

The distributed target environment for the SSP evolves over 30 years and has an indefinite period of operation. Furthermore, it encompasses processing resources on the ground, in low Earth orbit, in higher orbits and extending to the Moon. A standard interface set permits the software to be cost effective in meeting the needs of the application and to support processing which is safe and reliable with a heterogeneous collection of processors.

2.6 All objects and tools which are under configuration control within the SSE should be strongly typed. (As soon as possible, if not initially.)

Elaboration:

The benefits of strongly typed languages for building and maintaining reliable software are well documented. However this issue addresses the extension of those benefits throughout the life cycle support environment. For example, a set of requirements produced during the requirements analysis phase and currently under configuration control in the project object base should be under access control not only by personnel, but also by tool. In other words, for each object, there are known attributes and operations associated with its type. For a large, complex, evolving system such as the SSP, this strong typing can help prevent inadvertant or unauthorized creation, destruction or pollution of the object base. The proper use of the IRDS (described in 2.4) should greatly facilitate the implementation of such environments.

2.7 Most tools beyond the MAPSE should be methodology-based and should have a clear and consistent relationship to specific phases and/or activities of the life cycle. (A model is provided in section 4.0.) Formal interfaces should be defined for each software engineering activity and its supporting tools.

Elaboration:

The establishment of formal interfaces for each activity and its supporting tools permits retaining the benefits of evolving a comprehensive and consistent support environment (ie, an integrated environment) while importing the most important benefit of the open environment; ie, tools can be imported to support multiple methodologies for a given phase or activity as long as the formal interfaces are maintained.

2.8 Guidelines, methodologies and tool support to promote the development, importation, and application of reusable components should be a pervasive emphasis throughout all phases and activities of the life cycle of large, complex, distributed systems.

Elaboration:

Reusable components refer to any product of software engineering which can, with planning and management support, be cost effectively reused or adapted for reuse. Documentation templates and software templates are two examples which offer an enormous potential for lowering the cost of SSP software and improving its reliability. A modest investment in evolving an appropriate generic classification model which provides a standard interface to automated browsing and management tools is sorely needed.

2.9 Significant progress towards goals of:

. lowering life cycle costs

. lowering life cycle risks

. improving software quality

. improving human productivity

. improving the adaptability and reliability of software

can be achieved via the SSE in time to support the goals of the SSP only if we eliminate reinvention.

Elaboration:

There are two classes of reinvention that must be eliminated to the maximum extent possible, and as quickly and consistently as possible. They are: mistakes and solutions. The selection and use of appropriate models, standards and interfaces as discussed in the preceding list of issues and assumptions are believed to offer a major step toward the elimination of both types of reinvention

2.10 An Appropriate software engineering education and training plan for NASA and its constituency is essential to the success of developing and supporting both the SSE and the SSP.

Elaboration:

The preceding list of issues and assumptions are believed to be indicative of substantive trends and developments throughout many of the free world's leading institutions of government, industry, and academia. However, to leverage and to contribute to these advancements will require initiating and sustaining an appropriate education and training plan. Not only will this benefit the SSP, but it will also enhance the portability of tools, components, object bases, and people for other important projects.

3.0 An Outline of the Principal Requirements for a MAPSE

Table 1

Requirements to Architectural Features Relationship

Stoneman Requirement	Architectural Features
General Goals	
. Portability of user program	. Use of Ada and Retargetable Rehostable tools
. Portability of toolset	. Coding in Ada, layered architecture of environment
Database Requirements	
. Flexible repository for all project information	. File system model based on three node types with attributes and associates
. Every object in the database is accessed by its distinct name	. Ada-like path names and unique identifiers attribute
. Database shall permit relationships between objects to be maintained	. Associations
. Database shall support the generation control of configuration objects	. Revisions
. Selection within a version group	. By revision number
. Database shall support the generation and control of configation objects	. Attributes and associations manipulated by advanced configuration control tools
. Mechanisms shall be provided whereby objects needed to recreate a specified object will continue to be maintained	. Derivations history
. All objects connected with a specific project area can be grouped in a partition	. Hierarchical nature of the database
. General access control associated with a partition	. Node access control
. Support for program libraries	. Program libraries managed as directories in a database

Table 1 (con't)

Stoneman Requirement	Architectural Features
Database Requirements (con't)	
. Access control to any object in database	. Node access control
. database shall store information which allows management reports to be generated	. Predefined attributes, attribute access tools and command language processor
. Reliable storage of objects including long-term storage	. Backup and archiving tools
. Database must support differing "software configuration," both consecutive "released" and separate "models"	. Baselines defined by advanced configuration control tools and variation node in basic database structure
. History on configuration controlled objects must be maintained	. Advanced configuration control tools manipulating attributes and association of nodes
Control Requirements	
. A virtual interface which is independent of any host machine shall be provided	. Command language and KAPSE interfaces
. Achievability of command functions from Within programs	. Command Procedures written in command language or Ada programs using KAPSE services
. Command language is to be Ada	. Command language supported by a command processor which interprets the language
Toolset Requirements	
. Must support development of Ada programs, in particular separate compilation features	. Language processing tools and program libraries
. Tools shall be written in Ada where possible	. 90% of toolset in Ada
. Inter-tool communication shall be via the virtual interface	. All tools interface through command languages and/or KAPSE
. Uniform error handling	. Error status returns

Table 1 (con't)

Stoneman Requirement	Architectural Features
Toolset Requirements (con't)	
. Comprehensive "help" facility	. Help and help tools
. Extension to toolset supported	. Ada program access to KAPSE services and ability to execute Ada Programs directly from command language
. Testing and debugging of Ada programs at source level	. Debugger(s) and program analysis tools
. Resident testing and debugging facilities to host-resident	. Debugger and program analysis tools distribution between host and target
. The Minimal Toolset shall include - text editor - prettyprinter - language translator - linkers - loaders - set-use analyzer - control flow static analyzer - dynamic analysis tool - file administrator - command interpreter - configuration manager	. Toolset provide full support. These categories of tools are provided as individual tools, options within the compiler or as a collection of related tools

4.0 A Life Cycle Model for a Software Support Environment

The attached figure adapts the symbolic representation of McDermid and Ripken (1984) to the model described in DoD Standards 2167 and 2168 and to the major phases and activities to be supported by an integrated SSE.

The bottom of the figure depicts a foundation for the SSE beginning with a model for the "Life Cycle Project Object Base" (LCPOB). Ideally this model would be strongly typed and organized by phase attributes with traceability and accountability threads that can be clearly audited and enforced throughout the life cycle. The relevant schemas and dictionaries would be created and maintained via the EA/RA model in conformity with both the IRDS standard for schemas and dictionaries and the CAIS standard for interfacing and access control. The object base contains on-line objects and references to off-line objects which may be temporary or under baseline control. Included are libraries of technical tools, management tools, and reusable components.

The integrity and value of the LCPOB are maintained with the help of the tools and procedures of a hierarchy of three software engineering activities that permeate the life cycle: LCPOB management, configuration management, and quality management. Together with the LCPOB, these three layers of tools and procedures form a solid foundation for an extensible SSE that evolves over time to provide highly automated support for all phases of the project life cycle.

Within the phases (which are typically going to be subdivided into collections of activities), technical and management tools can be developed or imported which support the four components of the creating and recording functions common to each phase; i.e., the creation activities (shown as rectangles), the records they produce (shown as a closed pair of parallel arcs) the verification and validation of the effects of the creation activities based upon the records they produced (shown as circles) and the formal review process that allows the integration of management and technical perspectives (shown as a five sided figure). The top portion of the figure also depicts the milestone of acceptance testing which transitions software components from a development status to the maintenance and operation phase (shown as an oval).

LEGEND FOR: "A LIFE CYCLE MODEL FOR A SOFTWARE SUPPORT ENVIRONMENT"

Documentation

- o Symbol: A closed pair of parallel arcs.
- o Abbreviations:
  - . Sys RAD: System Requirements Analysis Document
  - . Sw RAD: Software Requirements Analysis Document
  - . Sw DSD: Software Design Specification Document
  - . Sw Des D: Software Design Documentation
  - . Sw Dev D: Software Development Documentation

Note: Other documentation not explicitly shown on the model but which is required by DoD Stds 2167 & 2168 and which should be supported by the environment include:

- o Life Cycle Content
  - . Computer Resources Life Cycle Management Plan
- o Management
  - . Software Development Plan
  - . Software Configuration Management Plan
  - . Software Quality Evaluation Plan
  - . Software Standards and Procedures Manual
- o Design
  - . System/Segment Specification
  - . System Requirements Specification
  - . Interface Requirements Specification
  - . Software Top Level Design Document
  - . Software Detailed Design Document
  - . Interface Design Document
  - . Database Design Document
  - . Software Product Specification
  - . Version Description Document
- o Test
  - . Software Test Plan
  - . Software Test Description
  - . Software Test Procedure
  - . Software Test Report
- o Operational Support
  - . Computer System Operator's Manual
  - . Software User's Manual
  - . Computer Support Diagnostic Manual
  - . Software Programer's Manual
  - . Firmware Support Manual
  - . Operational Concept Document
  - . Computer Resources Integration Support Document

LEGEND FOR: "A LIFE CYCLE MODEL FOR A SOFTWARE SUPPORT ENVIRONMENT"

Reuseable Components:

- o Includes documentation, requirements, specification, source code components, tools, applications libraries, test sets, budgets, plans, etc. In short, any work component with a potential to be developed, imported or purchased for cost effective reuse.

Verification & Validation

- o Symbol: Circle for V&V; five sided figure for formal reviews

- o Abbreviations:
  - . SRR: System Requirements Review
  - . SDR: System Design Review
  - . SSR: Software Specification Review
  - . PDR: Preliminary Design Review
  - . CDR: Critical Design Review
  - . TRR: Test Readiness Review
  - . FCA: Functional Configuration Audit
  - . PCA: Physical Configuration Audit
  - . FQR: Formal Qualification Review

Activity

- o Symbol: Three are shown as horizontal layers persisting throughout the life cycle:
  1) quality management
  2) configuration management
  3) project object base management

  Also note the symbols for the formal reviews and acceptance testing.

- o Definition: "The process of performing a series of actions or tasks."
  (Joint Regulation, SDS Documentation Set, 4 June 1985)

- o Note by CWM: Some activities extend across all phases of the life cycle (eg, the three listed above) whereas some phases contain a collection of phase-specific activities. (No such subdivisions are shown on this figure. However, these phase-specific activities also can be considered in terms of the four symbols common to the phases: activity, documentation, verification & validation, and review.)

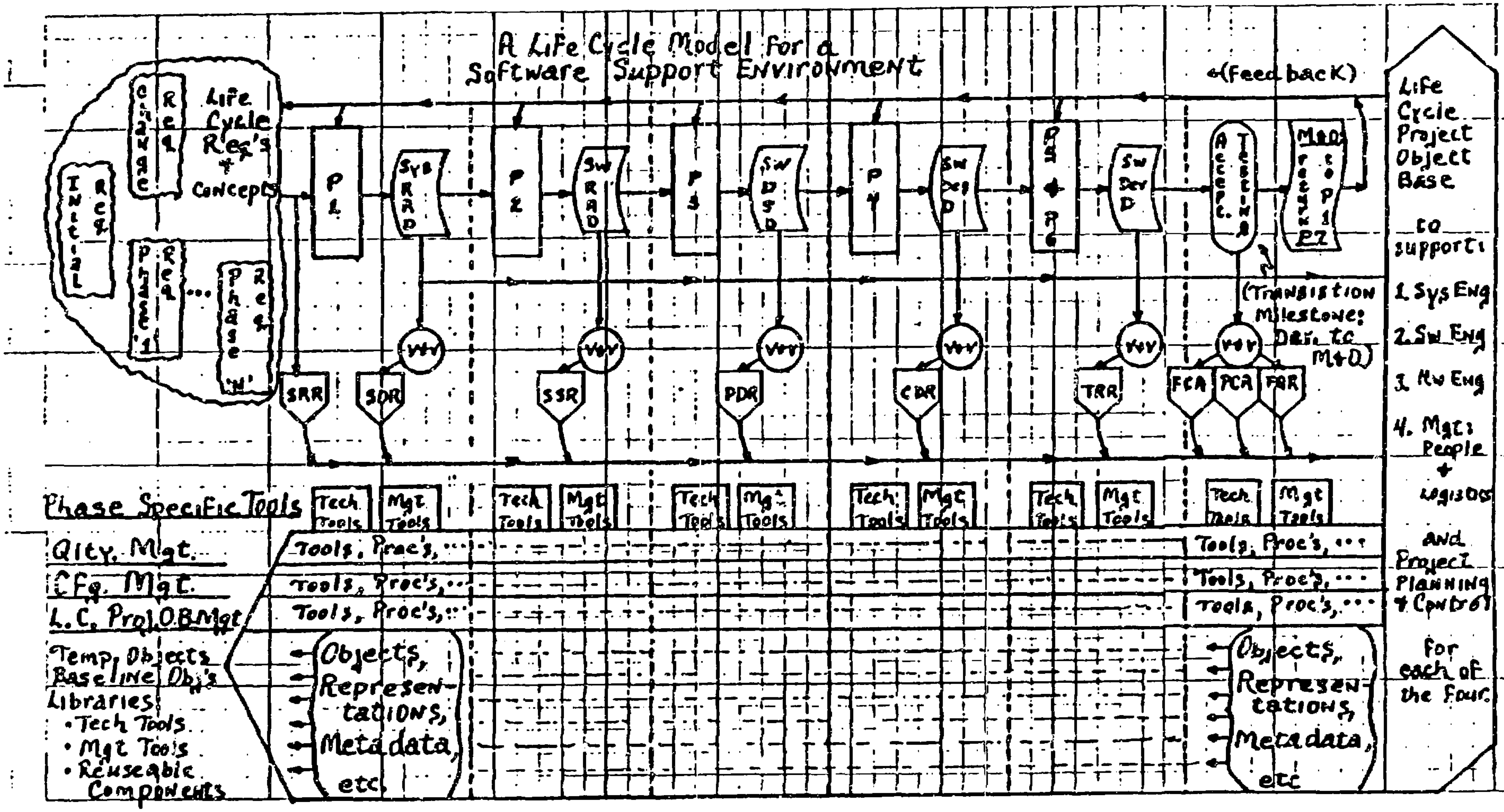
LEGEND FOR: "A LIFE CYCLE MODEL FOR A SOFTWARE SUPPORT ENVIRONMENT"

Phase

- o Symbol: Rectangle
- o Definition: "A discrete period of time delineated by a beginning and an ending event for each iteration in the incremental evolution of the life cycle." (CWM, 1985)
- o Abbreviations for 2167 Phases:
  - . P1: System Requirements Analysis
  - . P2: Software Requirements Analysis
  - . P3: Preliminary Design
  - . P4: Detailed Design
  - . P5: Coding & Unit Test
  - . P6: Computer Software Component Integration

Transition from Development to Maintenance & Operation

- o Symbol: Oval
- o Abbreviation:
  - . Acceptance Testing of Computer Software Configuration Items


A Life Cycle Model for a Software Support Environment
Initial Req
Change Req
Life Cycle Req's + Concepts
Phase 1 Req
Phase 'N' Req
SRR
P 1
Sys RAD
SDR
V+V
P 2
Sw RAD
SSR
P 3
Sw DSD
PDR
P 4
Sw Det D
CDR
P 5 + P 6
Sw Dev D
TRR
Accept. Testing
FCA
PCA
FQR
M+D: Post P1 P2
(Transistion Milestone: Dev. to M+D)
(Feed back)
Life Cycle Project Object Base
to support:
1. Sys Eng
2. Sw Eng
3. Hw Eng
4. Mgt: People + Logistics
and Project Planning + Control
for each of the four.
Phase Specific Tools
Tech Tools
Mgt Tools
Qlty. Mgt.
Cfg. Mgt.
L.C. Proj OB Mgt
Tools, Proc's, ...
Temp Objects
Baseline Obj's
Libraries:
• Tech Tools
• Mgt Tools
• Reuseable Components
Objects, Representations, Metadata, etc

5.0 A Prioritized List of Needed Tools for Initial Operating Capability (IOC) in June, 1987 and for the Operating Capability in June, 1990

Important Notes: all tools listed below are intended to extend and complement the MAPSE as described in Section 3.0 of this report. Furthermore, regardless of the development language the implementation is to be coded in, Ada should be used as an executable design language before development coding begins. Both configuration management and verification and validation are imposed upon the executable Ada design versions as well as the development code versions.

Legend :

R	= Required
O	= Optional
P1..P7	= Phases 1 through 7 of the Life Cycle Model
all	= Supports one or more activities throughout all phases of the life cycle

TOOLS	'87	'90	To Support
Problem Expression Editors	O	R	P1..P3
Syntax Directed/Template Driven Ada Editor	R	R	P3..P7
Semantics Language Processor	O	R	P1..P3
Semantics Analyzer	O	R	P1..P3
Semantics Information Browser	O	R	P1..P3
Dictionary & Schema	R	R	all
Report Generators	R	R	all
Reusable Components Browser	R	R	all
Reusable Components Design Aid	O	R	P4..P7
Modeling	R	R	P1..P4
Simulation	O	R	P1..P2
Resource Estimator	O	R	all
Automated Precedence Network	O	R	all
Schedule Generator/Analyzer	O	R	all
Change Request Tracker	R	R	all
Automated Work Breakdown Structure	O	R	all
Resource Scheduling Aid	O	R	all
Standards Checker	O	R	P4..P7
	O	O	P1..P3
Verifier/Assertion Analyzer	O	R	P4..P7
Prototyping	O	R	P1,P2,P4
Graphics Design Aid	O	R	P4
		O	P1..P3
Test Data Generator	R	R	P4..P7
Test Results Comparator	R	R	P4..P7
Test Harness	R	R	P4..P7
Scenario Generator	O	R	P4..P7
Environment Simulator/Stimulator	O	R	P4..P7
Performance Monitor	O	R	P4..P7
Black Box Test Generator	O	R	P4..P7
Data Extraction & Reduction	O	R	P4..P7
Test Completeness/Consistency Analyzer	O	R	P4..P7
Test Coverage Analyzer	O	R	P4..P7
Interactive Test Analysis (with a fully instrumented test bed)	R	R	P4..P7
Performance Model	O	R	P4..P7
Reliability Model	O	R	all
Event Signaling Path Generators	O	R	all
GKS Graphics			
2 dimensional	R	R	P4..P7
3 dimensional	O	R	P4..P7
Design Complexity Analyzer	R	R	P4..P7
Integrated Text & Graphics Forms Generator	R	R	all
Menu Manager	R	R	all
Command Language Script Manager for Distributed Processing	R	R	all
Word/Document Processing Integrated with Graphics Support and Electronic Mail and Filing across the Distributed System	O	R	all

TOOLS	'87	'90	To Support
Communications Tools to link the Hosts and Targets	R	R	all
Real Time Programming Aid	O	R	P4..P7
Expert System Generator	R	R	all
Cross Reference Analyzer	R	R	P4..P7
Statement Profile Generator	R	R	P4..P7
Compilation Order Analyzer	R	R	P4..P7
Intelligent, Automated Recompilation	R	R	P4..P7
Generic Instantiation Analyzer	R	R	P4..P7
Generic Instantiation Reporter	R	R	P4..P7
DIANA Tree Browser	R	R	P4..P7
DIANA Tree Expander	R	R	P4..P7
Impact Analyzer for Module Changes	R	R	P4..P7
Call Tree Analyzer	R	R	P4..P7
Elaboration Dependency Analyzer	R	R	P4..P7
Execution Metrics Analyzer	R	R	P4..P7
Run Time Support Dependencies Analyzer	O	R	P4..P7
Distributed Workload Simulator	R	R	P4..P7
Fault Tolerance/Safety Analyzer and Simulator	O	R	P4..P7
Fault Tolerance Programmers Aid	O	R	P4..P7
Run Time Support Environment Monitor	R	R	P4..P7
Run Time Support Timing Analyzer	O	R	P4..P7
Run Time Support Storage Analyzer	O	R	P4..P7
Run Time Support Tasking Analyzer	O	R	P4..P7
Target Network Topology Specifier	R	R	P4..P7
Target Node Resources Specifier	R	R	P4..P7
Distributed Target Node Restriction Checker	O	R	P4..P7
Partitioning and Allocation Tool	R	R	P4..P7
Distributed Program Builder	R	R	P4..P7
Distributed Program Resource Sharing/Replication Analyzer	O	R	P4..P7
Upgrade Load, Test and Integration Planning Aid for Non-stop Nodes	O	R	P4..P7

6.0 A Brief Description of the Tools in the Context of the Model

REQUIREMENTS ANALYSIS

CHARACTERISTICS, PRINCIPLES AND METHODS

Several approaches can be used during requirements interpretation, feasibility studies, and analysis.

Semantics Information Capture - Supporting interpretation, the capture of requirements in the form of a semantic model involves identifying key terms, categorizing the terms, defining the terms, and identifying the relations between the terms. The capture of semantics information creates a formal recording of the semantic model of the requirements, which becomes part of the baseline. Assuming the semantics information is machine-encoded, it might be expressed in a formal language such as Problem Statement Language (PSL) or in combination of formal graphics and text expression such as Software Requirements Engineering Methodology (SREM).

Semantics Analysis - Once the requirements are expressed in the context of a semantic model, the model relations can be used for a systematic analysis of the completeness and consistency of the requirements. This is achieved by asking questions which are answered with the aid of the relations, such as "Are there any other processes which should be related to Process A by the 'predecessor of' relation?"

Traceability may be established through reference relations between requirements and specification, design and code, etc. The relational analysis can be used to assess the impact of requirements changes on the baselined products.

The semantic analysis method also aids creation by identifying areas of requirements incompleteness or inconsistency.

Feasibility and Risk Analysis - Evaluating the feasibility of requirement is a significant part of requirements analysis. Feasibility should be viewed from the perspectives of design, performance and cost.

Design feasibility involves finding at least one design that satisfies the requirements. Any approach from trial design to prototyping is appropriate. Performance feasibility is a special case of design feasibility analysis. Once a trial design is established, modeling is an effective technique for analyzing performance. Cost feasibility involves estimating costs based on the trial design. Cost analysis must consider the three key elements: the development phase, the operations phase, and the phase for continuing adaptation.

SUPPORTING TOOLS

Several tools are effective in supporting the above.

Problem Expression Editors - These editors permit capturing a form expression of the problem in text and/or graphics form. The form expression obeys language-like rules and can then be machine processed v a Semantics Language Processor.

Semantics Language Processor - A language processor performs syntax checking and recording of semantics information entered via a form expression such as PSL. The results are available for later analysis or retrieval. The information resembles a data dictionary augmented by relations.

Semantics Information Browser - This tools stores and retrieves the semantic information. It might be based on a data base management system (DBMS) using relational techniques. An example of such a tool used in this application is IBM's Query-by-Example (QBE).

Semantics Information Analyzer - This tool uses the relational nature of the semantics information to perform consistency and completeness checks. Problem Statement Analyzer (PSA) is an example of a tool used to analyze data captured through the use of PSL.

Report Generator - General report generators would help create requirements analysis reports. Examples of reports are:

- Formatted problem statement report, which gives the original relationships in a well-structured format,
- Structure report, which presents relationships as a hierarchy,
- Data structure report, which lists data structures and the types of their contents,
- Data/activity interaction, which shows interaction between data objects and activities, and
- Picture report, which diagrams the direct relationships of an object.
- 2167/2168 reports

Modeling Tool - This tool provides queuing theory aids tailored to descriptions of computer systems. The tool assists in developing performance analysis models. Performance Oriented Design is an example of such a queuing tool. Other modeling tools besides queuing would also be appropriate.

Resource Estimator - An estimation model helps assess cost feasibility. The model uses characteristics of the software system and the development approach to predict required manpower, schedule, and computer resources.

Proto:yping/Simulation - These are software tools for developing simulatic or computer based prototypes of software systems. They could consist of a high-level language and an interpreter, with statistical analyses packages The U.S. Air Force tool, SAINT, is an example.

REQUIREMENTS ON THE SSE

The requirements on the SSE data base, derived from requirements analysis, are:

Baselined Products for Past and Present Phases of Evolution - The semantic information is the only data associated with requirements analysis thε should be baselined. It should be under configuration control and subjec to change only as requirements changes are approved. Baselined data shou not only include the "shalls" of each phase (which must be dichotomousl demonstrated at acceptance test time) but also the "shoulds" which hav life cycle implications that cannot be dichotomously demonstrated a acceptance test time and which may require the design of special metric and instrumentation to support their analysis at subsequent points in tl life cycle.

Non-Baselined Data - Any information associated with modeling, simulation, prototyping, or semantic analysis should be saved temporarily. It should be used later in requirements analysis iteration or other activities.

Measurement Data - Several measurements of the requirements analysis activity and its outputs should be captured:

- Size of the data base for semantics information,
- Complexity of the requirements as measured by the relationships in the semantics information and
- Number of inconsistencies or omissions found.

PRELIMINARY DESIGN/DESIGN SPECIFICATION

CHARACTERISTIC, PRINCIPLES AND METHODS

Formal Recording

The specification information must be recorded in some suitable form. As a minimum, the specification should describe:

Translation - o: the "shalls" from requirements analysis into Ada packa specifications. Functional requirements should be transformed in functional Ada specifications that can be checked by an Ada Compile Non-functional requirements (i.e., constraints) should be transformed in a discipline of Ada comments that can be checked by other APSE tools.

Interactions - the interactions of the software system and its outside environment (i.e., descriptions of the I/O device interfaces, sensor interfaces, etc.);

DETAILED DESIGN

CHARACTERISTICS, PRINCIPLES AND METHODS

Three groups of design support are identified: formal recording of system design, formal recording of data and program design, and creative aids.

There are several techniques for recording the system design.

Information-Hiding - This technique involves isolating information within modules. The module limits are defined by the information (design decisions, data definitions, etc.) to be isolated. Design is based on the expected changes to the information, thus localizing the effect of future changes.

Module Specification - This technique allows others to determine the intent of a complete module by reading the module specification.

Uses Hierarchy - This technique explains which programs depend on the correct implementation of a given module to produce correct results.

The techniques for the formal recording of data design and program design are:

Program Design Language (PDL) - PDL is a useful technique for formally recording the program design. It is sufficiently low-level to support direct coding, and is flexible enough to leave some questions unanswered while the designer proceeds with the design. (i.e., Ada sour code with Ada "stubs".)

Stepwise Refinement - This technique goes hand in hand with PDL. With stepwise refinement, specifications for the lower level code become part of the documentation of the procedure. This makes the intent of the code much clearer.

Abstraction of Data Types - With abstraction, the designer can develop details where they are needed. This permits information-hiding as well as a more independent implementation of the system.

Many creative techniques exist for design. A designer chooses techniques based on their individual approach to creativity. Some prefer graphic techniques while others do not. The choice of creative techniques should be left to the individual, whereas the techniques for formal recording must be standard. Described below are some representative creative aids:

Data and Control Flow Analysis - Module decomposition and functi allocation are based upon the data and control flows required by t system. An example is
Structured Design.

Data Structure Transformation - Transformation is a design technique in which the structure of the input and output data determines the structure of the program.

Graphic Decomposition Techniques - Graphs showing hierarchic relations depict the decomposition at many levels. An example is Structured Analysis and Design Technique (SADT).
Graphic Control Descriptions - Other ways of showing the control flows in the program are Petri Nets and Warnier-Orr diagrams.

SUPPORT TOOLS

Design/Specification Language Processor - These check syntax and connections. Input is the system design and the module specifications written in some standard syntax. Output is a list of inconsistencies or syntax errors. (eg, an Ada compiler for functional requirements a: other MAPSE tools for non-functional requirements.)

PDL Syntax Analyzer - These detect mismatched interface items and force the designers to maintain a consistent syntax for the design. Input is a design written in a PDL, and output is a list of syntax errors and inconsistencies in data usage. An Ada compiler supports this.

PDL Interpreter - These allow a design to be executed before it is actually coded. The interpreter's input is a design written in a PDL with program input data. Its output shows the result of applying the input data to the design. An Ada compiler supports this.

Graphics Package - These support the creative phase of design. Various designs could be displayed graphically to aid the designer in making a choice. Input might be module descriptions, hierarchy relations, or data-use relations. Outputs include graphic representations of this information.

Modeling Tools - These are used for judging how feasible particular designs are. A model might take a high-level design as input. It then produces execution and timing statistics. These statistics are used to determine if the design could meet the performance requirements.

Report Generators - These transform stored design into documents and reports. Thus the baselined design will be stored in machine-readable form, permitting required documents to be produced easily.

REQUIREMENTS ON THE SSE

As with the other activities of development, the data base must contain information on the design.

Baselined Products - Throughout the life of the system, the most recently approved form of the design must be stored in the data base. The system design are entered before the design of various subsystems or modules.

Non-Baselined Data - This includes preliminary designs as well as graphic displays used during the creative process. Graphic displays include tree structures, block diagrams, and other material created by design tools. The data base must provide for maintaining the temporary designs developed before one is actually chosen and baselined.

Measurements - These should include module interconnection measurements, such as data bindings. These should also include lower design measurements, such as cyclomatic complexity, and operators and operands. Many of these measurements are normally taken on the completed code, but with good, low-level PDL, they can be taken (or approximated) during design.

Archival Data - Archived data should capture the motivation behind the choice of design. The archived data should also include past designs evolved from use or rejected during development along with the reasons for the rejection.

VERIFICATION AND VALIDATION

CHARACTERISTICS, PRINCIPLES AND METHODS

The methods linked with correctness analysis are either static analysis or dynamic analysis. Static analysis includes, in order of increasing rigor, reviews, inspections, and proofs of correctness. Dynamic analysis includes all testing techniques.

Reviews - Reviews determine the internal completeness and consistency of system requirements and software specification, design and test information. They also assess its consistency with its predecessor information. Reviews involve a broad range of people, including developers, managers, users, and outside experts or specialists. A review must have specific objectives and questions to be addressed. The review findings generate rework tasks for the development group.

Inspections - Inspections evaluate the correctness of component level specification, design, code, test plans, and test results. They are more formal and rigorous than reviews. An inspection involves a small group of people of a specific make-up, and follows a well-defined procedure.

Proofs of Correctness - All development products should be verified with an informal proof of correctness. Certain critical kernals of code or special applications may require a formal proof of correctness.

Testing - Dynamic execution of the system or system component with known inputs in a known environment is a "test". If the test result is consistent with the expected result, the component is deemed correct in the limited context of the test. The following baselined documents are created relative to testing:

- Test Plan - Defines the scope, approach, and resource needed for testing.

- Test Procedures - Provides a detailed description of the steps and test data associated with each test case.

- Test Results - Documents the results of each test run. Unsuccessful runs trigger trouble reports which must be addressed by the development group.

There are two approaches to testing--black-box testing and white-box testing. Black-box testing uses only knowledge of externals (to the function) while white-box testing uses knowledge of the internal design of the function.

Black-box testing uses the specification to develop test cases and is mo appropriate for system testing because it directly demonstrates that the implemented system satisfies the specification. White-box testing uses design information to develop test cases and is most appropriate for component testing.

The relationships between system functions and component or system test cases should be clearly established. Then, when changes are made to parts of a system, a subset of test cases can be identified which will test the system sufficiently. This process is called regression testing. Effective regression testing is a good way to reduce software development costs.

SUPPORTING TOOLS

A representative set of tools to support validation is listed below.

Performance Model - Performance models evaluate system performance constraints such as response time. The evaluation is performed by an analytic model or simulation model based on the system design. If the performance model is used to evaluate requirements feasibility, a high-level design is assumed. Performance models have been developed for many systems. Currently developed is a generalized modeling tool called Performance Oriented Design.

Prototyping Aid - Developing an operational prototype allows evaluation of requirements and design approaches. Executable design language are useful in this regard.

Consistency/Completeness Analyzer - These tools aid analysis of internal consistency and completeness when specification requirements are expressed in a machine-interpretable form.

Standards Checker - Standards checkers perform static analysis of code or documentation and identify standards violations.

Verifier/Assertion Analyzer - These analyzers verify that code correctly implemented specifications by checking the truth of the assertions (embedded in the code) against actual program execution.

Symbolic Execution - Symbolic execution is used for proving the correctness of certain classes of programs. It involves "executing" them symbolically to prove certain assertions.

Theorem Prover - Theorem provers automate proof-of-correctness techniques.

Test Harness - A test harness provides the framework for unit testing of procedures. With it, programmers can interactively define the procedure interface, prepare test data, run an instrumented test, and display the test result.

Test Data Generator - Based on a given testing strategy such as "test every path", this tool will automatically develop test cases based on t code or perhaps the design.

Hardware Simulator - Hardware simulators allow object code for a target computer to be tested on the host computer.

Host-Target Compiler - A software alternative to hardware simulators which use source text to generate object code that executes on the host computer.

Test Results Comparator - This tool compares actual unit test results against expected results. It issues a trouble report if the test results are not correct.

Environment Simulator/Stimulator - The simulator duplicates the operational environment of the ECS in a test facility. This is done before integration with the sensor and effector systems. The stimulator tests the system by simulating input such as sensor and effector interface signals.

Performance Monitor - These tools permit measurement of system performance parameters, such as response time, algorithm processing time, channel utilization, etc.

Data Extraction and Reduction - These tools analyze the system dynamically. During execution, data is captured and stored, and later reduced by processing to create reports for the dynamic analysis activity.

Scenario Generator - These tools control the complex parameters which create the scenarios and drive the simulation of the environment

Black-Box Test Generator - These tools generate functional test skeletons by using the specification of the software system. The analyst then completes the scenarios by adding detail to the skeleton. The tools should include sampling techniques to assure adequate coverage of the specification.

Test Coverage Analyzer - These tools analyze the testing status of Software Component Item as software progresses from unit testing integration.

Interactive, Fully Instrumented Test Bed - These tools support interacti testing between the host and a target test bed that is fully instrumente The environment stimulator/simulator may be used to provide a context f the subsystem component under test.

Reliability Model - These tools use the error history of the system over its life cycle to estimate a reliability measure for the system.

REQUIREMENTS ON THE SSE

The requirements on the SSE data base, derived from the correctness analysis, are summarized below:

Baselined Products - Test plans, test procedures and test results (of correctly executed tests) are all baselined. They are controlled by configuration management. The results of inspections and proofs might also be baselined.

Non-Baselined Data - The non-baselined data includes work-in-progress, static analysis data, trouble reports, and debug data. Temporary storage of this type of information is required.

Measurement Data - A number of measurements associated with correctness analysis should be captured. These include: number of modifications to a unit, number of errors found per unit, number of test runs, number of errors by error category, and test coverage.

PROJECT MANAGEMENT SUPPORT

CHARACTERISTICS, PRINCIPLES AND METHODS

Estimation - Most resource estimation techniques use the measurements from prior projects to estimate resources. Support of estimation methods requires a data base of comprehensive measurements including such software system parameters as size of source code, source language, development resources expended, and complexity measures.

Precedence Networks - This planning method is used to analyze task dependencies and to determine the critical path of development activities. Such an analysis is usually needed to define a realistic schedule. It is also useful in evaluating contingencies and creating contingency plans.

Change Control - This is the core of configuration management. It controls all changes to baselined products. The approval process for changes might be as follows:

- The written request for change is submitted to the configuration management function. It might come from a change in requirements or from a trouble report documenting a defect.

- An assessment is made of the technical feasibility of the change, and its impact on schedule and budget. If it has t potential to endanger life and property, a separate safe assessment may be made.

- The change is approved or disapproved based on its potenti effect upon safety, its value and its cost.

- The development plan is modified and resources adjusted to add approved changes.

- The fully verified change is entered into the new baseline.

SUPPORTING TOOLS

Many tools are applicable to management and can support the methods and activities described above. Some of the more useful tools for management include:

Resource Estimation Model - This software uses a data base of measurements on past projects, along with a description of the new system, to create resource estimates for development.

Automated Precedence Network - This software creates precedence network charts and determines the critical path based on the input of detailed milestones and precedence relations.

Automated WBS - This tool helps to create budgets and a work breakdown structure.

Schedule Generator - This tools uses output from the precedence network, and organizational responsibilities (related to the WBS), to create schedules by organizational entity.

Change Request Tracker - This tool logs change requests when submitted, tracks them through the approval cycle, and records their resolution.

Resource Scheduling Aids - These tools permit resources, such as computers, conference rooms, terminals, and test equipment, to be scheduled. Usage reports on the resources and the scheduling of the resources are the main functions of these tools.

Event Signaling Path Generators - These tools allow events such as fork a join points in the precedence network or the schedule generator or even in the resource usage reports to be linked with communicatio messages/notices to be automatically forwarded to designated individua when the event occurs.

Report Generators - Report generators create management reports on technical, budget, and administrative status.

REQUIREMENTS ON THE SSE

The activity of management imposes the following requirements on the SSE data base.

Baselined Products - The development plan, although not a part of the software system or its descriptive information, should be maintained as a baselined product to insure proper management of changes to the plan. Configuration management data and quality assurance plans should also be baselined.

Non-Baselined Data - Significant amounts of information associated with the management must be kept temporarily. This information includes engineering change requests, trouble reports, resource allocation plans, actual resource utilization reports, technical milestone status, action item status, and the results of quality assurance reviews.

Measurement Data - Many measurements are of interest to management. These include the number of engineering change proposals (ECP), and trouble reports (TR), time to process an ECP or TR, resource use for each ECP or TR, resource use by project activity, and software size and complexity measures.

UTILITIES WHICH SUPPORT ONE OR MORE ACTIVITIES THROUGHOUT ALL PHASES OF THE LIFECYCLE

CHARACTERISTICS, PRINCIPLES AND METHODS

There are many utilities which are useful to both management personnel and technical personnel throughout the lifecycle. Even in those cases where one group is responsible for creating as well as reading the information, and the other group is primarily a consumer of the information, these utilities are useful for both groups to understand. They include such things as dictionary and schema definers, report generators, reuseable components browser, schedule generators, electronic mail and filing, and ad hoc report generation.

SUPPORTING TOOLS

Dictionary and Schema Tools. All large, complex systems can be divided into subsystems. The structure of the subsystems is usually defined by the technical personnel as a schema. The definitions of the subsystems elements are typically stored in a dictionary. The IRDS draft standard proposes a standard way of defining and maintaining the schema and dictionary with the EA/RA model.

Report Generators. Both scheduled and ad hoc reports are prepared by both technical and management personnel. Wherever possible, report templates should be generated and stored as reuseable components. DOD standard 2167 contains several examples of such templates.

Reuseable Components Browser. Although many components are potentially cost effective to reuse throughout the lifecycle, management can make effective use of reuseable documentation components which may include plans, budget formats, etc. Technical personnel can make effective use of both reuseable software and reuseable documentation components. A browser which provides semantic information overlayed upon a common classification scheme with key words can facilitate this reuse.

Resource Estimator. Several microscopic tools for estimating resources required for a small activity, to macroscopic tools for estimating resources required for a subsystem are needed by both technical and management personnel.

Automated Precedence Network. This is a similar tool to one described under project management support.

Schedule Generator/Analyzer. This is similar to the one described under project management support.

Change Request Tracker. This is similar to the one described under project management support.

Automated Work Breakdown Structure. This is similar to the one described under project management support.

Resource Scheduling Aide. This is similar to the one described under project management support.

Reliability Model. This is similar to the one described under verification and validation.

Event Signaling Path Generators. This is similar to one described under project management support.

Integrated Text and Graphics Forms Generator. This is useful to both management and technical personnel for generating new forms appropriate for use on a specific portion of the project.

Menu Manager. This allows both technical and management personnel to create their own menus for personal or team utilization.

Command Language Script Manager for Distributed Processing. Many projects require both technical personnel and management personnel to access computing resources across a collection of distributed hosts. For example, a report is to be generated requiring access to objects at three remote sites with the results to be printed at a fourth site. This tool allows those scripts that are to be used frequently to be prepared, stored and scheduled for appropriate distributed processing.

Communications Tools to Link Hosts and Targets. Both management and technical personnel are likely to need host-to-host communications. Technical personnel are also likely to need host-to-target communications for such purposes as interactive debugging, status queries, upgrades, etc.

Word/Document Processing Integrated With Graphic Support and Electronic Mail and Filing Across the Distributed System. Electronic mail and filing are useful to both technical and management personnel across the distributed system. This support can be particularly useful if it can be integrated with graphic support and an interface to the project object base and other utilities.

REQUIREMENTS ON THE SSE

Baselined products - The products accessed and manipulated by these tools which are under baseline control include: dictionary's, schemas, and certain schedules which are formally approved parts of the development plan.

Non-Baselined Data. Much of the information associcated with the tools of this section must be kept temporarily. This may include memos, ad hoc reports, certain forms, and scripts of the dialog between host and the working subsystem of a target environment.

Measurement Data. Many measurements are of interest to managment. Of particular interest are schedule changes and changes to the dictionaries and schema.

CODING, UNIT TESTING AND INTEGRATION TESTING/DEVELOPMENT

CHARACTERISTICS, PRINCIPLES AND METHODS

Designs which map program entities across distributed processing resources should be specified in two complementary parts. First, the functional requirements should be demonstrated to be met by the program design by executing the program in the host environment. (Ie, compile and execute the Ada source code on the host system without regard to properties of distribution.) Second, the non-functional requirements (ie, constraints) such as the location each program entity is to be assigned, timing constraints, sizing constraints, etc. should be mapped to a simulator for analysis of the implications of imposing these restrictions upon the design which was proven in the first step. Tuning of assignments, code, algoithms and structures can take place in the host environment until the simulator provides a degree of confidence. Load modules can then be built and moved to the target environment or to a target test bed for further study. The implementation should produce an effective, understandable transformation of the design. The automatic generation of appropriate comments in the source code can ease the more complex process of maintenance in a distributed environment.

The following are some key aspects of implementation:

Standard Interface Set to a Catalog of Runtime Support Environment Features and Options. This interface set establishes a virtual Ada machine. The compilation system produces target code that uses the services provided by the standard interface set. The requested services determine which modules of the run time support library are to be exported to the target environment.

Target Network Topology Specifier. This allows the designer to specify the symbolic names for remote area networks, local area networks, and individual processing nodes. The design also identifies the communications support available to link the various entities of the network.

Target Node Resources Specifier. This tool allows the designer to specify the hardware resources for each node identified with the network topology specifier. The system will retain this information in the project object base along with the collection of software resources that will be assigned to this node later in the design. The designer declares the instruction set architectures available, the memory banks and their attributes, the buses and their attributes, and the communications links that are available.

Partioning and Allocation Tool. After the Ada source code has been transformed into a DIANA representation and executed to demonstrate that it meets the functional requirements of the program, a discipline of comments and key words such as "location" can be used to map each program entity to a symbolic location. This symbolic location corresponds to those node and network identifications previously entered with the topology specifier and the node resource specifier. These non-functional requirements are added as attributes to the DIANA representation.

Distributed Workload Simulation. After the symbolic location assignments and other constraints have been added to the attributes of the DIANA representation, the workload simulator examines the project object base to determine characteristics of the already existing workload (if any) and to select empirical estimates of communication delays, processing throughput, and other relevant estimators. A simulation is then provided for analysis. If the analysis indicates the design approach is not feasible, new approaches to distribution can be provided by returning to the preceding step.

Distributed Program Building. When the workload simulation indicates a feasible design, the process of building new load modules includes examining the symbolic location assignments added to the DIANA tree and looking these up in the project object base to determine what type of instruction set architecture the particular entity's object code is to be generated for. If the code is to be added to the workload of an existing system, it is also necessary to identify if additional modules or new versions of the run time library need to be added or if additional hardware is likely to be needed to accomodate the increase in workload. The end result of the program building activity is to prepare a load module consisting of applications code and the necessary support from the run time library for each of the processors affected by the distribution of the program entities.

Run Time Support Environment Monitoring. If life and property are to depend upon the program meeting both its functional and its non-functional requirements, it may be desirable to prepare the program for execution in a target testbed. To be effective, the testbed should be fully instrumented and interact with the host environment. This requires the support of a run time monitor for each processor in the target testbed to interact with the instrumentation and the host environment to provide meaningful information.

Other implementation activities are similar to those encountered in a system which generates object code for a single embedded processor.

SUPPORTING TOOLS

Target Network Topology Specifier. This tool permits the topology of remote area networks, local area networks and individual processing nodes to be identified along with the communication paths linking them together.

Target Node Resources Specifier. After the nodes were identified via the target Network Topology Specifier, this tool allows the processors, memory resources, bus resources, IO resources, and communications paths available to each node (plus their relationships and attributes) to be specified. This system will automatically add and maintain a list of the software resources assigned to each target node. This information is maintained under configuration management control in the project object base.

Partitioning and Allocation Tool. This tool uses a discipline of Ada commenting to add symbolic information concerning the non-functional requirements of the program (such as location assignment) to be mapped to the entities of an Ada program.

Distributed Workload Simulator. This tool reads the symbolic location of assignments and other constraints as attributes in the DIANA representation. It selects estimators of the impact of the non-functional requirements from the project object base, and executes a simulation of the program to permit analysis.

Distributed Program Builder. This tool examines the attributes added to the DIANA representation by the partitioning and allocation tool and uses this symbolic information to look up the target node resources in the project object base. From this information it selects the appropriate back end code generators for the processors that will be affected by the distribution of the program entities. It also checks to see if additional modules are needed from the run time library. A load set is then generated for each processor.

Run Time Support Environment Monitor. This tools executes in the run time environment of a processor either in a target testbed or the actual target system. It captures and reports information relevant to the testbed instrumentation and/or the interactive host environment. The information is generally useful in studying the behavior of the system.

Run Time Support Dependencies Analyzer. This tool identifies the run time library modules that will be required by the application object code. It indicates the probable impact on size and throughput based upon frequency and type of constructs and referencing from a table of empirical data that is improved over time.

Run Time Support Timing Analyzer. This tool works with a run time monitor to provide a more accurate perception of the timing requirements of the run time kernel, the run time library modules, the applications code, communications, and other overhead.

Run Time Support Storage Analyzer. This tool works with the run time monitor to provide insight into the manipulation of both statically and dynamically determined storage requirements. The information is particularly useful in evaluating program units for their storage utilization. It can provide summaries and warnings for inappropriate utilization.

Run Time Support Tasking Analyzer. This tool evaluates tasks for interactions with other tasks creating static tasking profiles. This can be used with run time monitoring in detecting deadlock and other tasking interaction failures. It can also be used in analyzing the static scheduling of tasks as well as their dynamic performance.

Distributed Target Node Restriction Checker. In an incrementally evolving system, many target nodes acquire additional or changed workload assignments over time. Because of restrictions placed by the processor design on available memory and because of other restrictions such as representation specifications which assign certain IO devices to particular addresses, this tool checks to see if the additional workload proposed for the node will conflict with these restrictions.

Distributed Program Resource Sharing/Replication Analyzer. Frequently, communications overhead can be reduced and throughput increased by deliberately replicating many resources in a distributed program (eg, constants, common processing routines). This tool assists the designer in determining the options and the potential effects of resource sharing versus replication.

Upgrade Load, Test and Integration Planning Aid for Non Stop Nodes. Frequently, a large complex distributed network such as the SSP will require a number of non-stop services. Unattended satellites, unattended free flying platforms, and, often, attended locations where life and property depend upon a continuous provision of the sevices, will require interactive testing with the host environment. Upgrades and reconfiguration should be able to take place dynamically without bringing the system to a stop. While the experts know how to accomplish these functions, a planning aid is useful to prepare for the transition. This tool serves as such an aid.

DIANA Tree Browser. This tool is useful in supporting the maintenance of other tools such as the partitioning and allocation tool and in supporting the evolution of additional tool functionality.

DIANA Tree Expander. This tool complements the browser and provides for adding/modifying DIANA representations.

Fault Tolerance/Safety Analyser and Simulator. Fault tolerance and software safety are of particular concern in multiprocessor and distributed applications. Fortunately, not all subsystems and not all programs require the extra overhead associated with fault tolerant software. However for those programs that do require fault tolerance and software safety analysis, this tool can be in-valuable.

Fault Tolerance Programmers Aid. This tool assists the programmer in developing fault tolerant programs for distributed, multiprocessor, and single processor applications.

Real Time Programmers Aid. This tool aids the designer in designing hard scheduled, real time programs to map to distributed, multiprocessor, and single processor applications.

Expert System Generator. This tool assists the programmer in generating experts systems that can co-exist with the applications code in distributed, multiprocessor, and single processor applications.

Impact Analyzer for Module Changes. This tool allows the designer to ask "what if" types of questions. Specifically, the impact of changing an interface specification or of changing the definition of a private type can be quickly identified.

Analyzer for Elaboration Dependencies. The Ada language rules allow for partial elaboration to be ordered. This can be particularly troublesome when the elaboration takes place across a distrubuted collection of computing resources. This tool analyzes these effects.

Compilation Order Analyzer. This tool reads any collection of source files and generates a report of the order of required compilation.

Intelligent, Automated Recompilation. The Ada language rules state that any changes in the public specification of a package may cause a need to recompile those programs dependent upon the package. However if the change is to add a new function without modifying any of the existing functions or interfaces, then many of those modules that use this package may not have to be recompiled. Similarly if the change is to a single function that was not used by many of the other modules, recompilation can again be minimized. This tool provides automated recompilation when the rules are obvious. It notifies the user when there are questions requiring human judgement.

Call Tree Analyzer. This tool produces a report showing all resources that a given program module calls and is called by. Both direct and indirect calls are indicated with a brief description of each reference.

Execution Metrics Analyzer. This tool provides relative statistical analysis of the execution of a program in the host environment.

Cross Reference Analyzer. This tool provides the standard cross reference listing for one or more program units. All references to declarations in a program unit from anywhere in the program space are shown.

Statement Profile Generator. This tool profiles the utilization of selected Ada constructs in an identified program module.

Generic Instantiation Analyzer. This tool allows the user to locally instantiate a version of the generic reuseable component and to test the version with defined test values. If the locally instantiated version works as expected, the user then feels more secure in utilizing the instantiation inside his source code modules.

Generic Instantiation Reporter. This tool reports on all generic reusable components that have been instantiated since the module was added to the reusable components library. It also reports other relevant data about the forms of the instantiations.

Reusable Components Designers Aid. This tool assists the designer in designing, developing, verifying, and documenting components to be proposed for validation and inclusion in the reusable library.

REQUIREMENTS ON THE SSE

The most important requirements and opportunities for the SSE life cycle project object base become evident from this phase. The results are summarized below:

Baselined Products. The functional requirements are similar to those described in the preceding sections. However, opportunities arise dueto the requirements for the DIANA representation in the implementation phase. An estimated ten to twenty times the processing time is required to convert Ada source code to DIANA representation as compared to converting the DIANA representation to object code for the the target environment. Furthermore source code and object code can both be reconstructed from the DIANA representation. Since the Stoneman requirements for the MAPSE provides a unique identification for each object produced (which includes history attributes identifying the time, date, tools, etc. used to manipulate the object), an enormous amount of on-line storage space can be conserved in the project object base if the DIANA representation is maintained in the baseline.

The other important implication for baseline control as a result of this phase is the identification and maintenance of the network topology and the network node resources described in the preceding tools and methods.

Non-Baselined Data. The temporary storage required for this category is similar to the functional requirements listed in the other sections of this report. However, the savings and storage space made possible by the utilization of a DIANA representation described above may be significant even for temporary storage requirements.

Measurement Data. A number of metrics regarding the utilization of these tools is desirable. Knowing who is using the tools for what projects, and knowing the frequency of reference can provide valuable management insights.

NAVSEA
PMS - 408

ALSN: Multiprogramming
Distribution

CONTROL
DATA

MULTIPROGRAMMING

and

DISTRIBUTED Ada*

* Ada is a registered trademark of the U.S. Department of Defense (AJPO).

4/86 KITIA\DP_01

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL DATA

DISCLAIMER

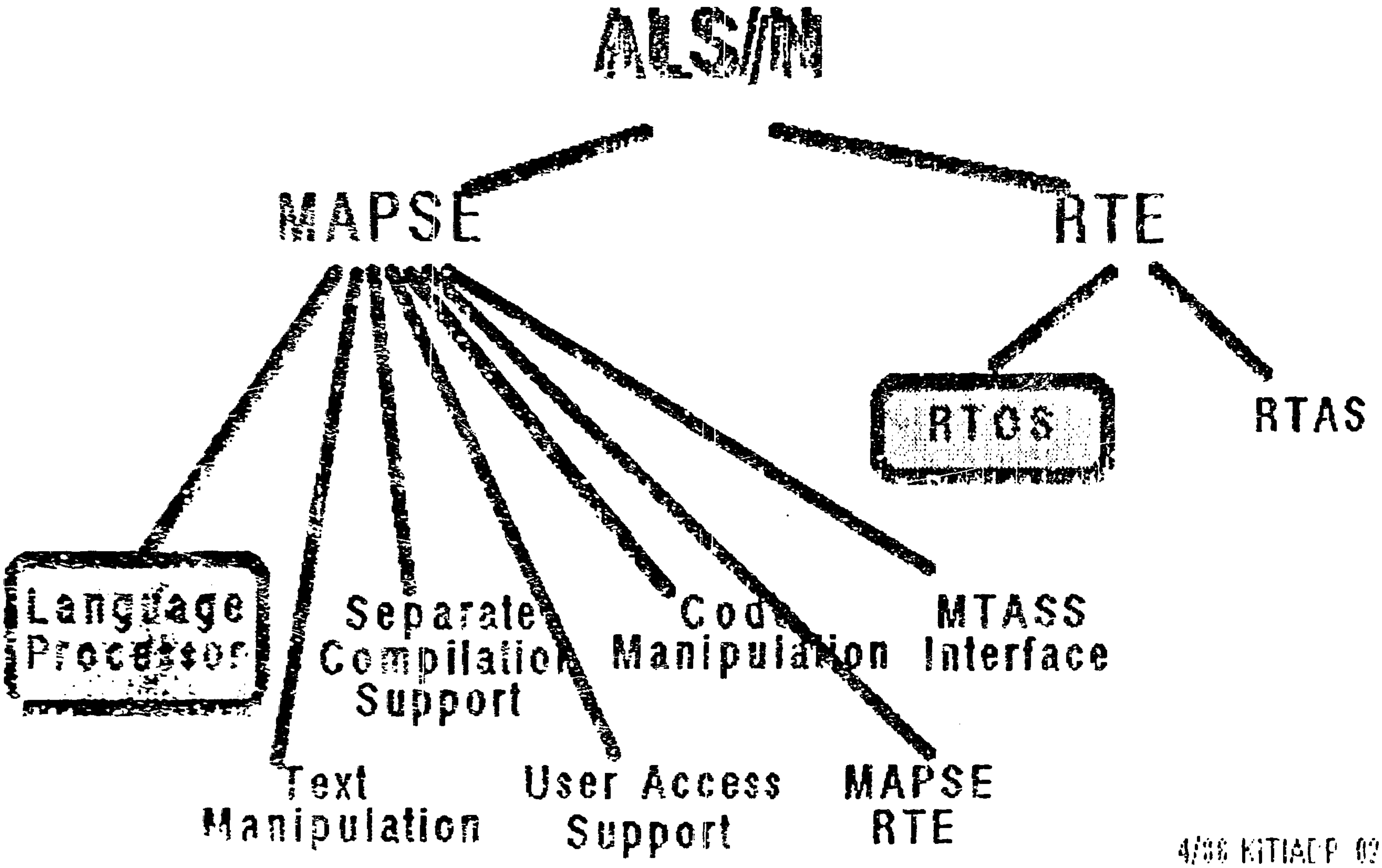
The approaches to multiprogramming and distributed Ada contained herein are NOT currently approved by the Navy or PMS - 408. Directions will be determined as a result of 'lessons learned' in build 1 of Ada/L and Ada/M.

4/86 HITIA\OP_01B

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA



4/86 KITIACP_02

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL DATA

MULTIPROGRAMMING

Intraprogram Control

- Creation
- Deletion
- Suspension
- Resumption

Communication

4/86 KITIA\DP_03B

NAVSEA
PMS - 408

ALSN: Multiprogramming
Distribution

CONTROL
DATA

MULTIPROGRAMMING:

SINGLE TIERED SCHEDULER

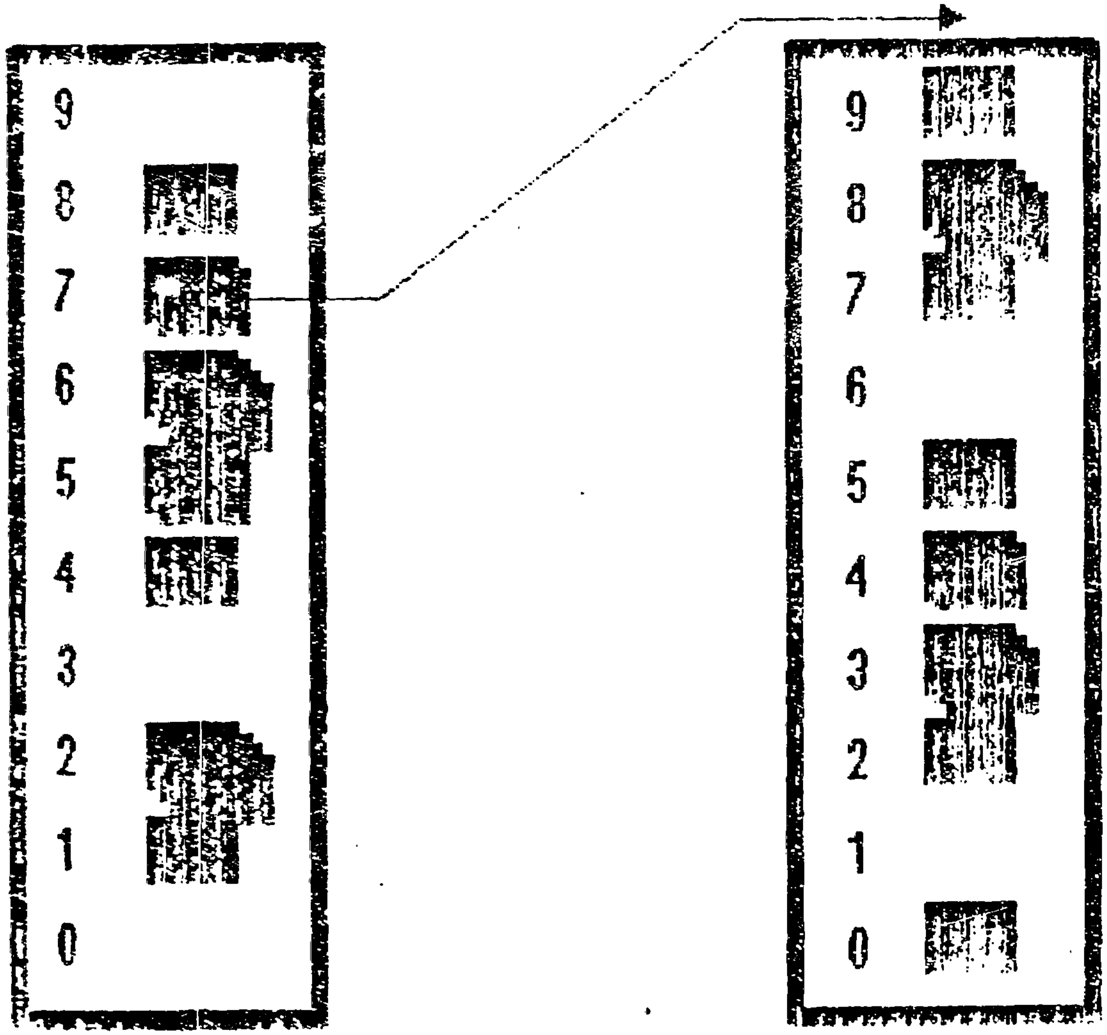
- Appears like single program to Scheduler
- Relative priority determined dynamically.

4/86 KITIA\DP_03

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

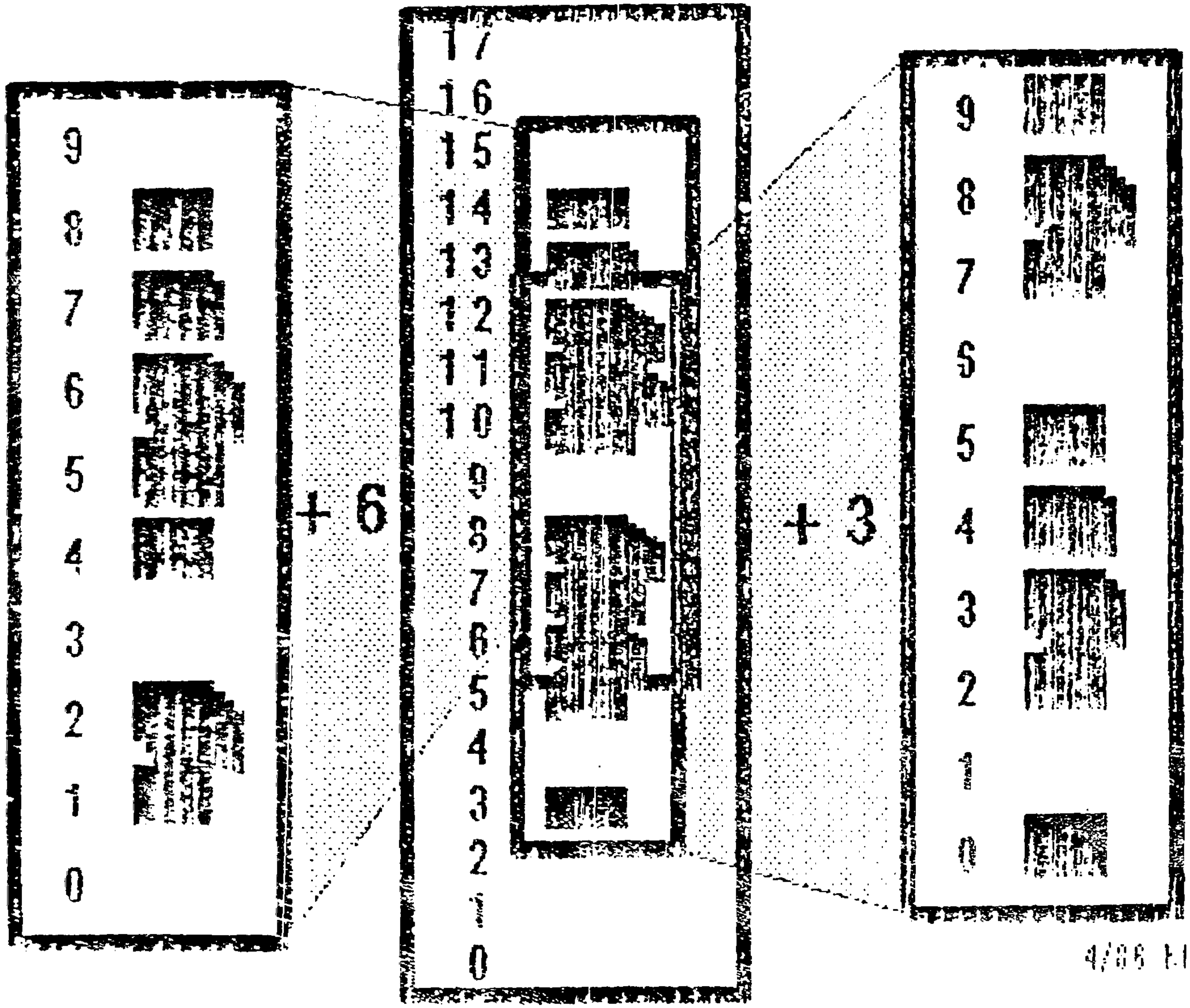


4/86 KITIA\DP_04

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA



4/86 HITIA\DP_05

DISTRIBUTED PROCESSING

a different approach

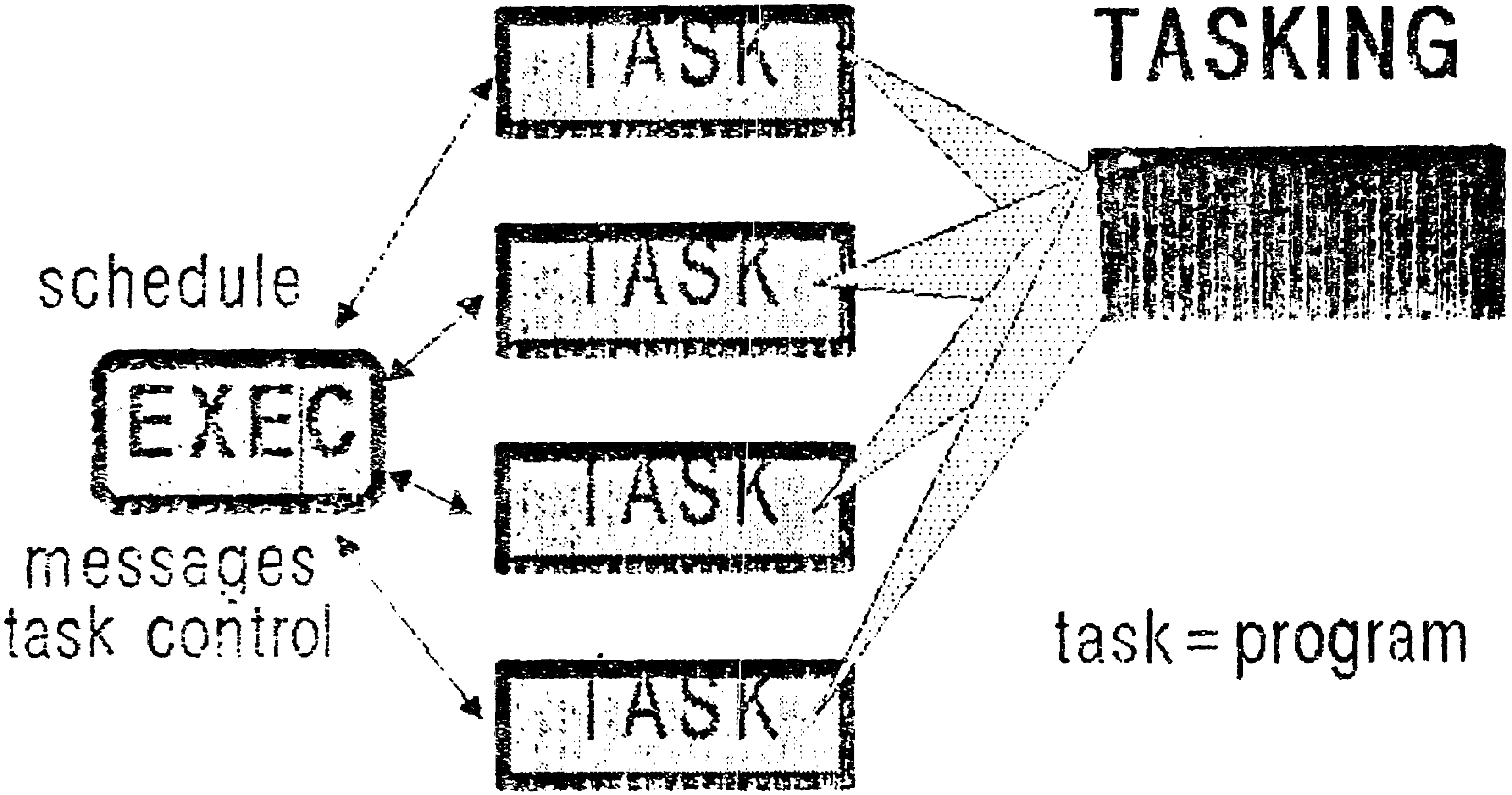
4/86 KITIADP_06

NAVSEA
PMS – 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

Pre – Ada TASKING



task = program

4/86 KITIADP_07

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

PRE – Ada Distributed Processing

Distribute on TASKS (programs)

EXEC complexity

level of tasking transparency
tightness of apparent coupling

Programs tend to need internal knowledge of distribution.

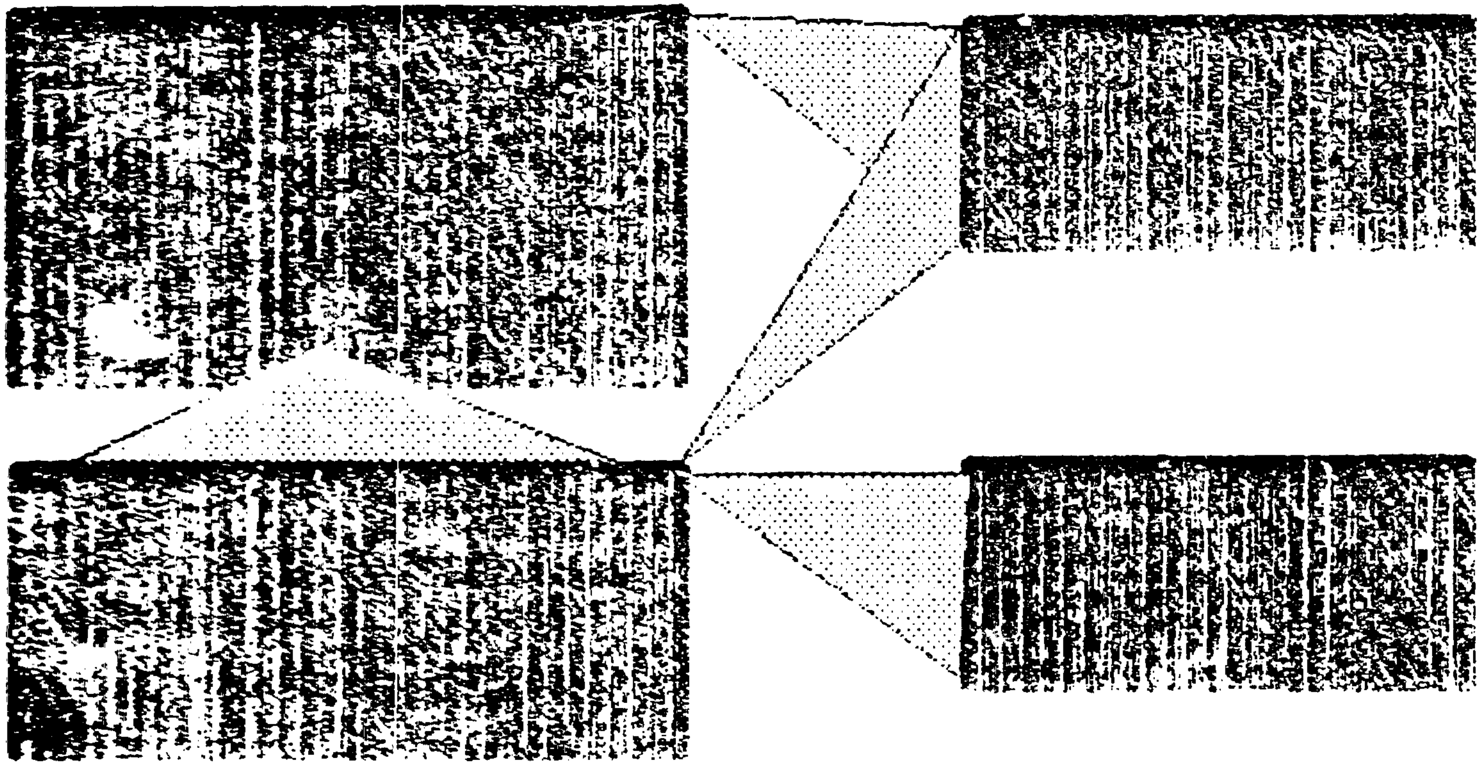
4/86 KITIA\DP_98

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

ADA TASKING



THERE IS NO NATURAL WAY TO DISTRIBUTE

4/86 KITIA\DP_09

ALSN: Multiprogramming Distribution

Distribution on Task Boundaries

Would require an extremely complicated RTExec to satisfy Ada semantics.

OR

Would require significant constraints on the use of the Ada language.

4/86 KITIA\DP_10

NAVSEA
PMS - 408

ALSN: Multiprogramming
Distribution

CONTROL
DATA

Distributed Ada Exec: PROBLEMS

- How to call a procedure of librar; unit in another machine.
- Handle dependent Tasks
- Visibility rules not amenable to partitioning
- Intra-machine Rendezvous (distributed Entry queues)

4/86 KITIA\DP_11

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

Key:

There is no knowledge on one machine of the detailed state of any other machine.

4/86 NITIA\DP_12

NAVSEA PMS - 408	ALSN: Multiprogramming Distribution	CONTROL DATA

SO HOW DO WE ACHIEVE DISTRIBUTION?

WHAT ARE THE UNITS OF DISTRIBUTION?

4/86 KITI43DP_13

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

UNIT OF DISTRIBUTION: **LIBRARY UNIT**

Simplifies Scope Rules:

- All cross machine access is to objects identified in a Package Spec

Simplifies Tasking:

- Task object always located with master.
  (with proper allocator rules)
- Termination rules simplified.
- Entry queues can be localized.

4/86 KITIA\DP_14B

NAVSEA
PMS - 408

ALSN: Multiprogramming
Distribution

CONTROL
DATA

OTHER CHARACTERISTICS:

- Static Configuration
- Distribution Specified at Compile TIme
- Distribution Transparent to Program

Build 1 introduces problems similar to those present in distributed situation.

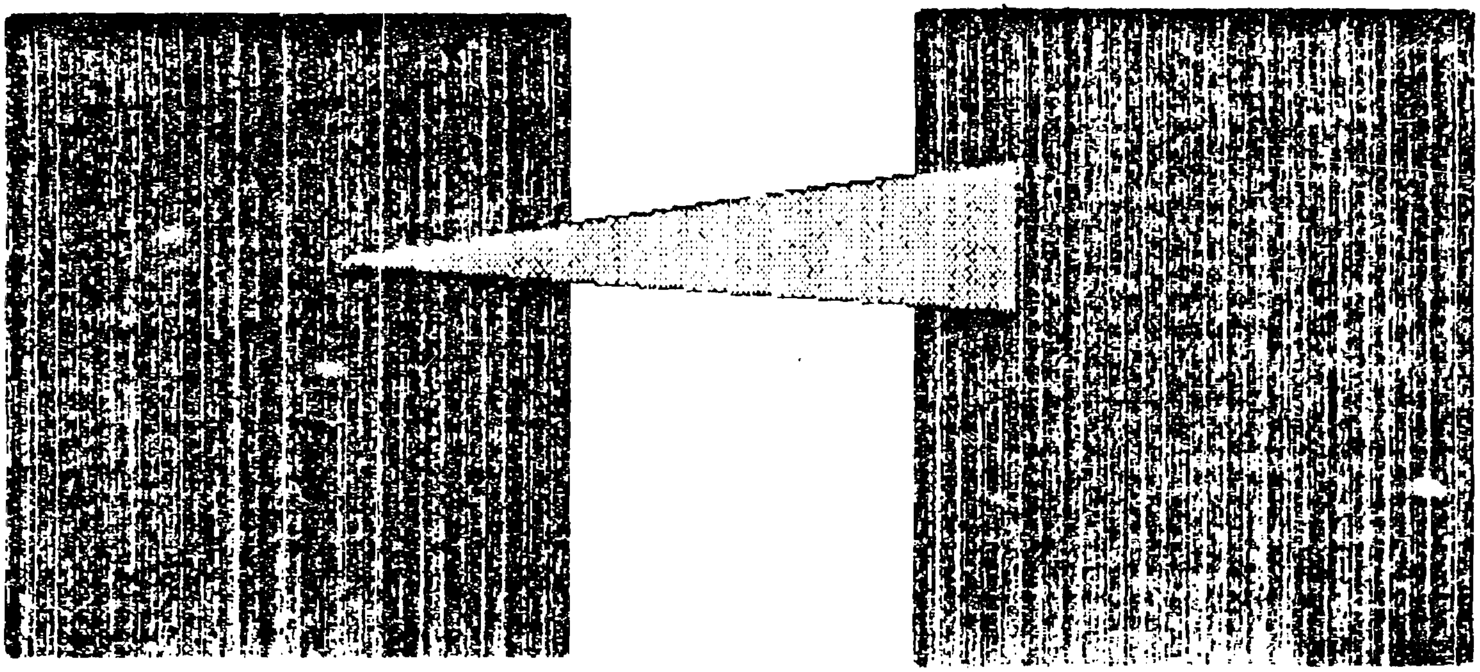
4/86 NITIA/DF_15

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

DIGRESSION: Procedure Call



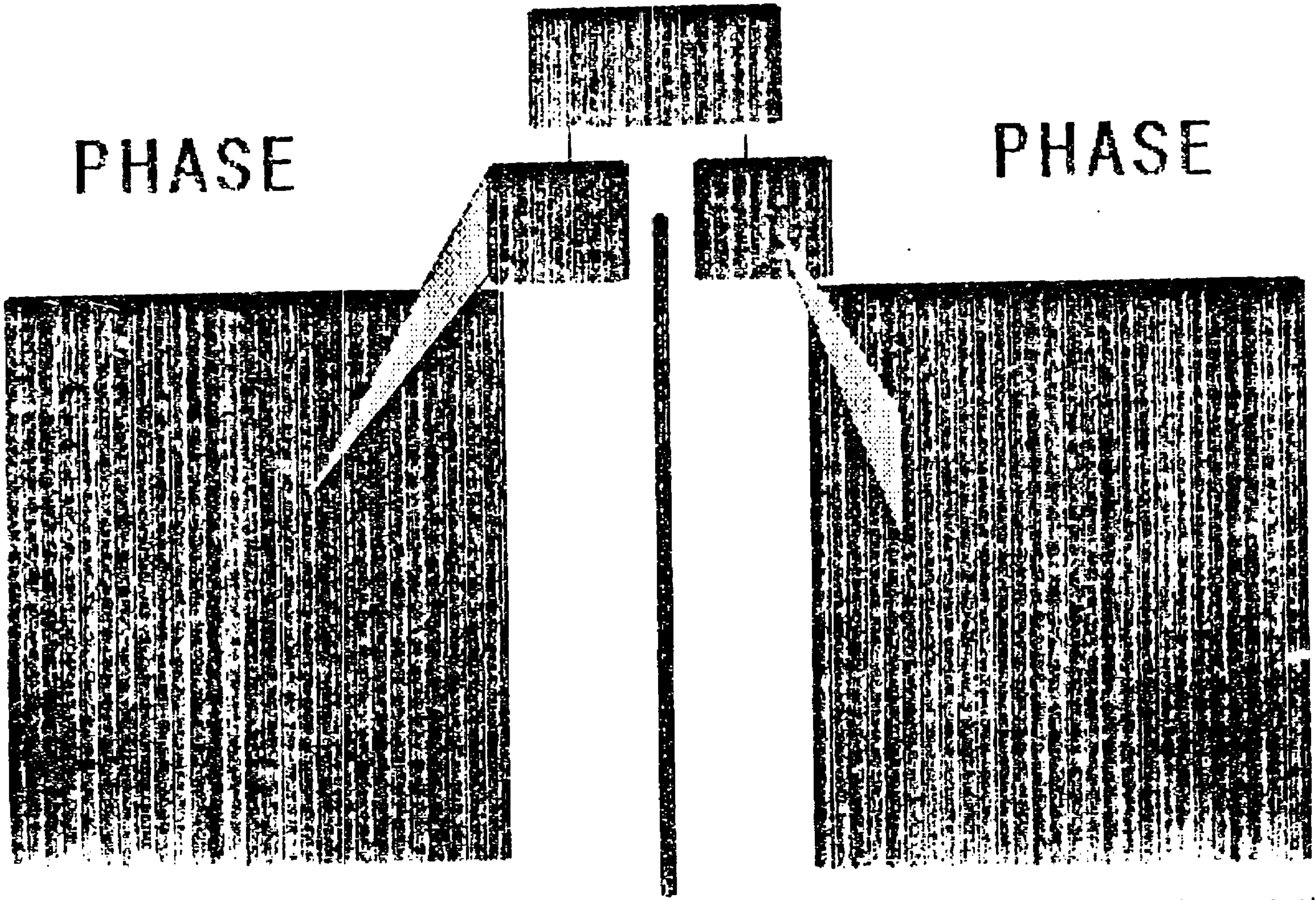
but what if not enough memory reach for both packages?

4/86 NITIA\DP_156

NAVSEA
PMS-408

ALSN: Multiprogramming Distribution

CONTROL
DATA



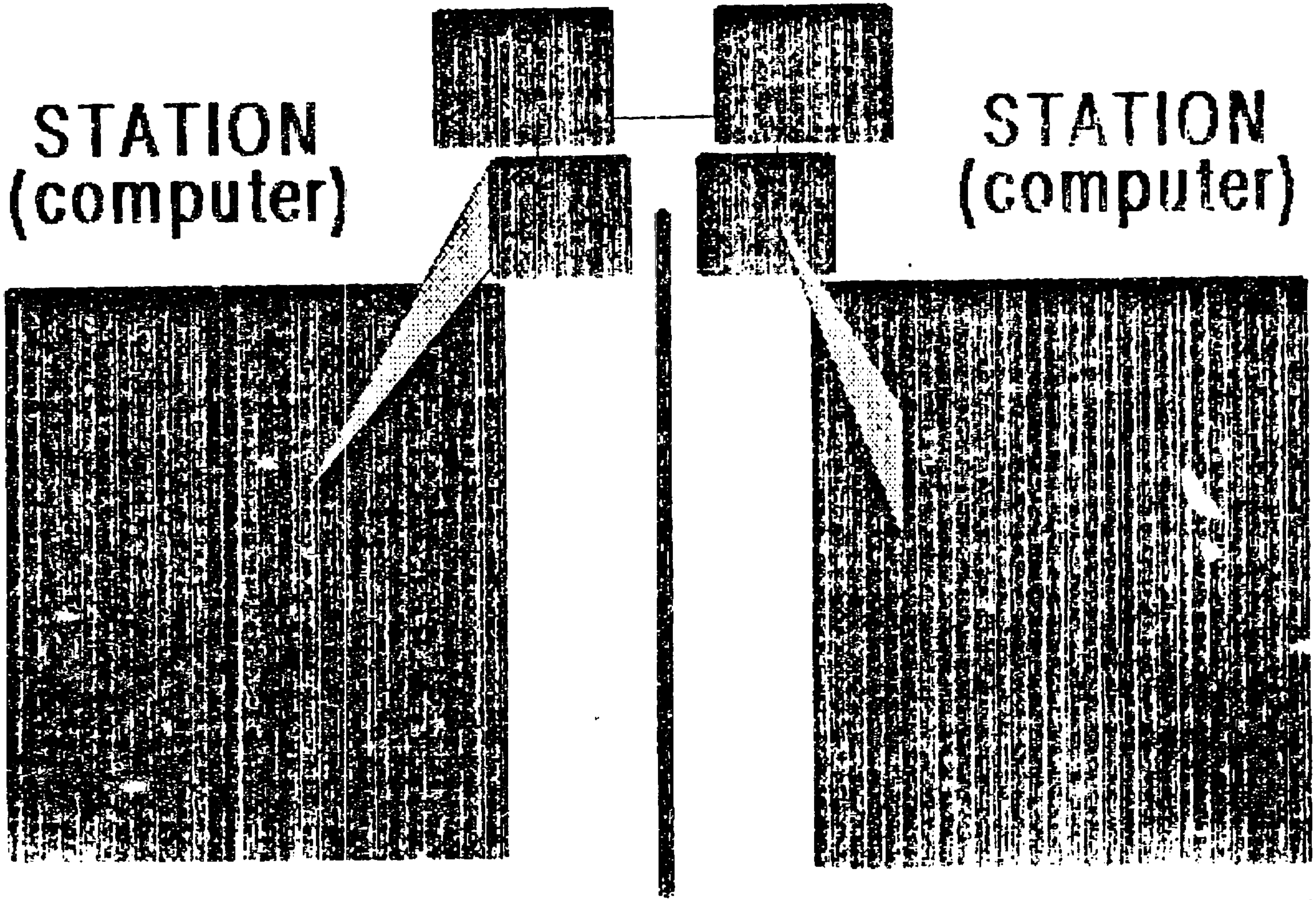
4/86 KITIA\DP_15C

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

Remote Procedure Call



4/86 KITIA\DF_15d

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

CHARACTERISTICS:

- Knowledge of distribution contained in program.
- Remote Procedure Call is general mechanism.
- Access to shared data through compiler inserted procedure.

WHAT ABOUT TASK ENTRIES

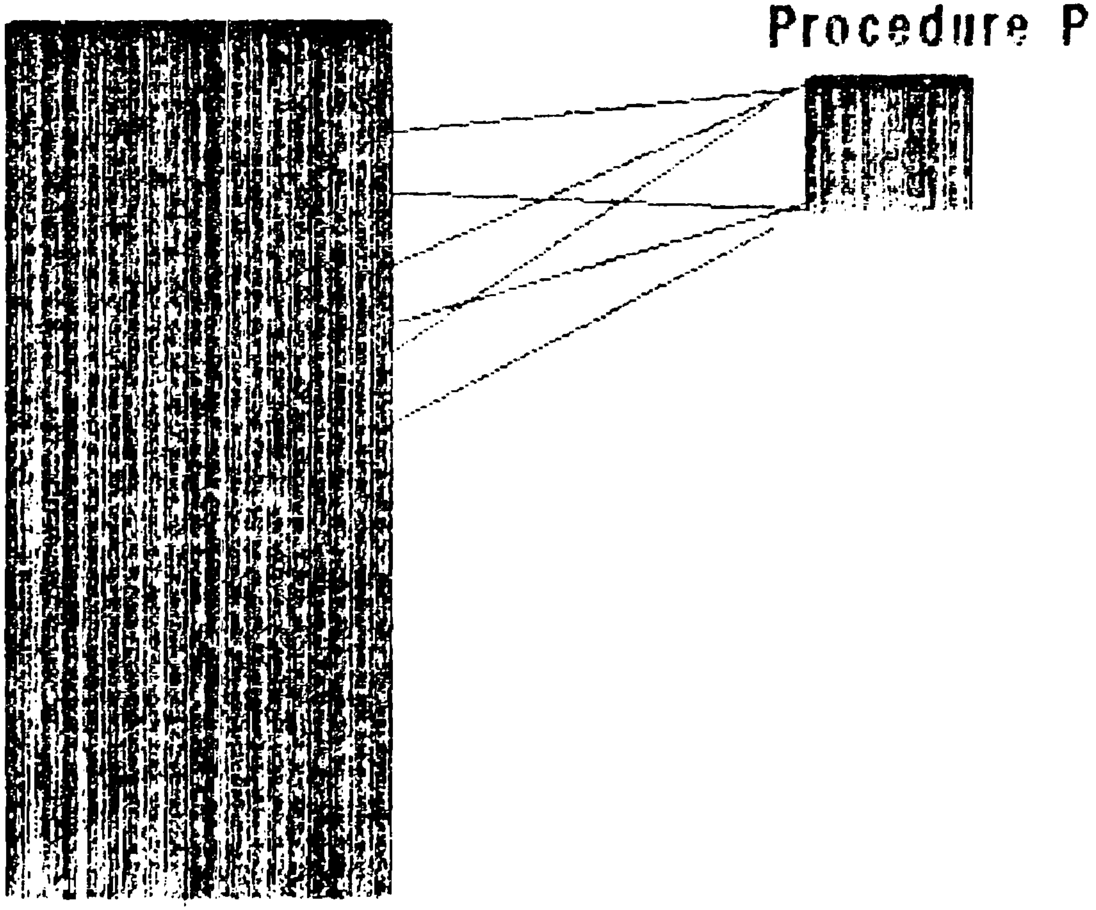
4/86 KITIA.DP_16

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

PRAGMA INLINE(P);



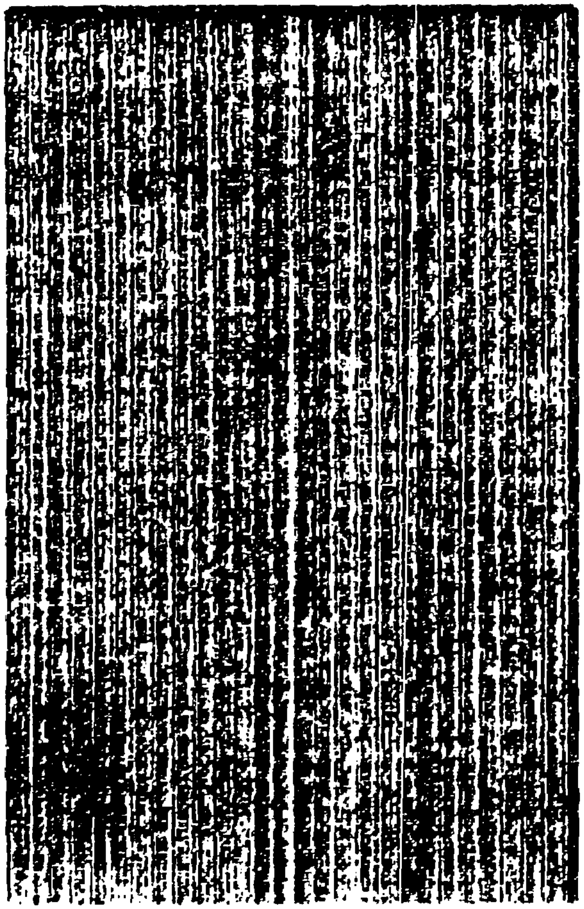
Procedure P

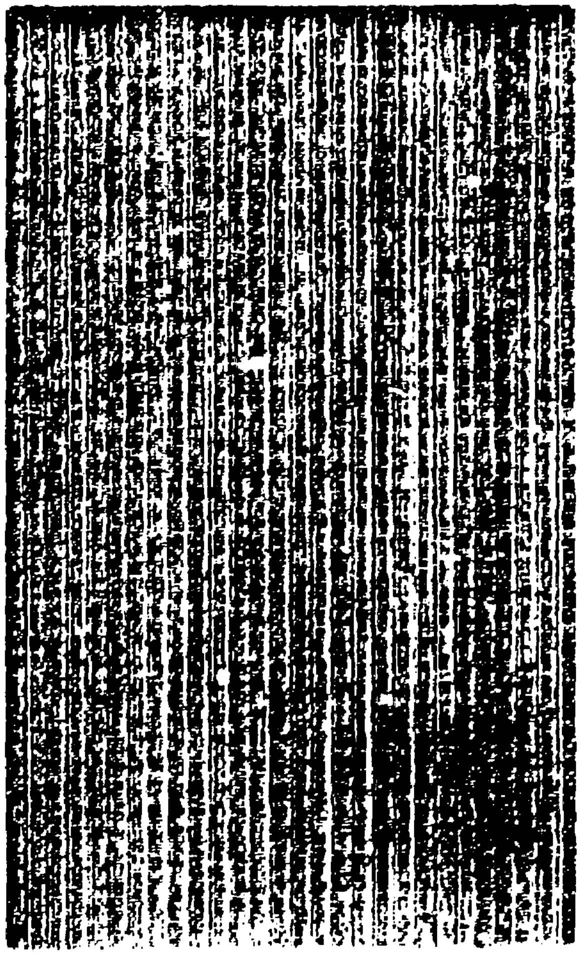
4/88 MITIADE_17

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA





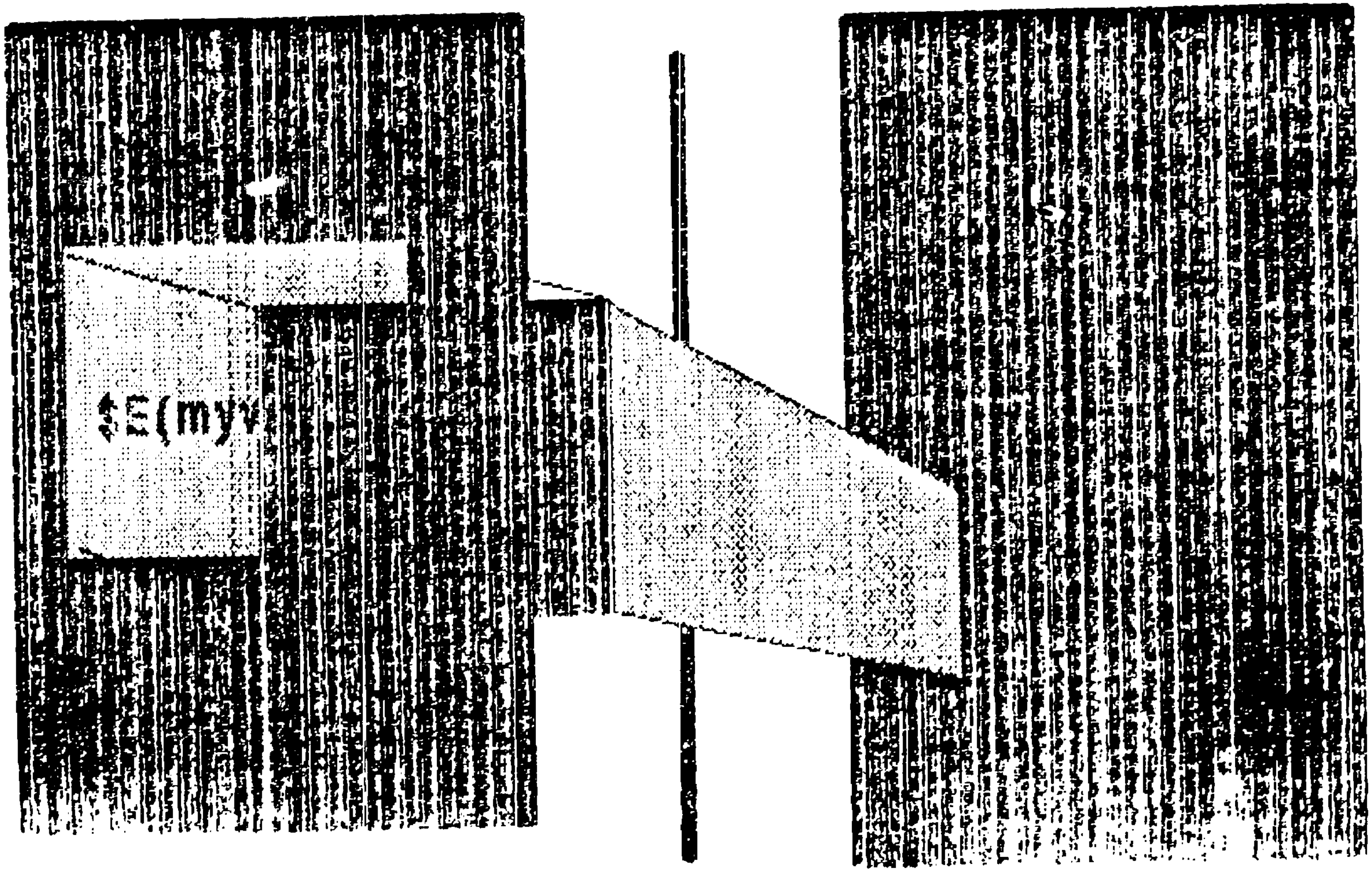
4/86 KITIA\OP_18

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

OUTLINE



4/86 EITIAD°_198

TASK RENDEZVOUS:

Cross machine entry calls are "outlined" and moved to where primary thread of control is.

Procedure call substituted.

Remote procedure call is implementation mechanism.

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL DATA

TASK RENDEZVOUS:

Since all tasking interaction takes place on the machine containing the master task obj.

The RTExec is not distributed but rather can be viewed as a local exec with a fancy comm system.

4/86 VITIA\DP_20

NAVSEA
PMS - 408

ALSN: Multiprogramming
Distribution

CONTROL
DATA

SHARED DATA:

Multiple Copies of objects are created in equivalent lexical positions in each remote station to the master copy of the object.

The value of the copies are updated at sychronization points.

4/86 NITIALIP_21

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

DESIRE THAT APPLICATION BE

INDEPENDENT

OF ITS PARTITIONING ONTO THE

DISTRIBUTED SYSTEM.

4/86 KITIA\DP_22

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

APPLICATION SYSTEM INDEPENDENCE IS ACHIEVED THROUGH SEPARATION OF PROGRAM SOURCE FROM PARTITIONING INFORMATION.

4/86 KITIA\DP_23

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

THIS NEW LANGUAGE IS CALLED

ADA PROGRAM
PARTITIONING LANGUAGE

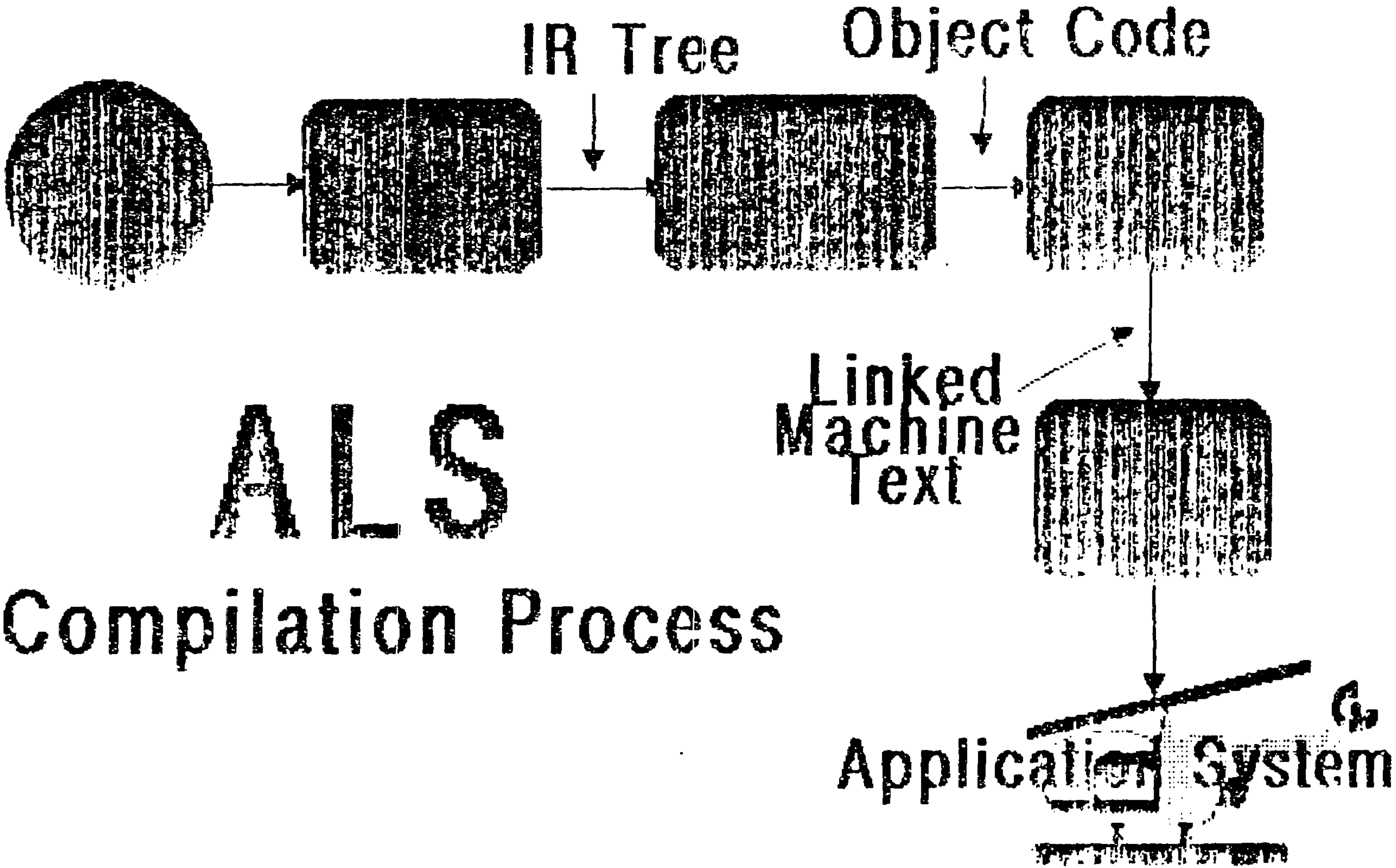
APPL

4/86 KITIA/DP_24

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA



ALS

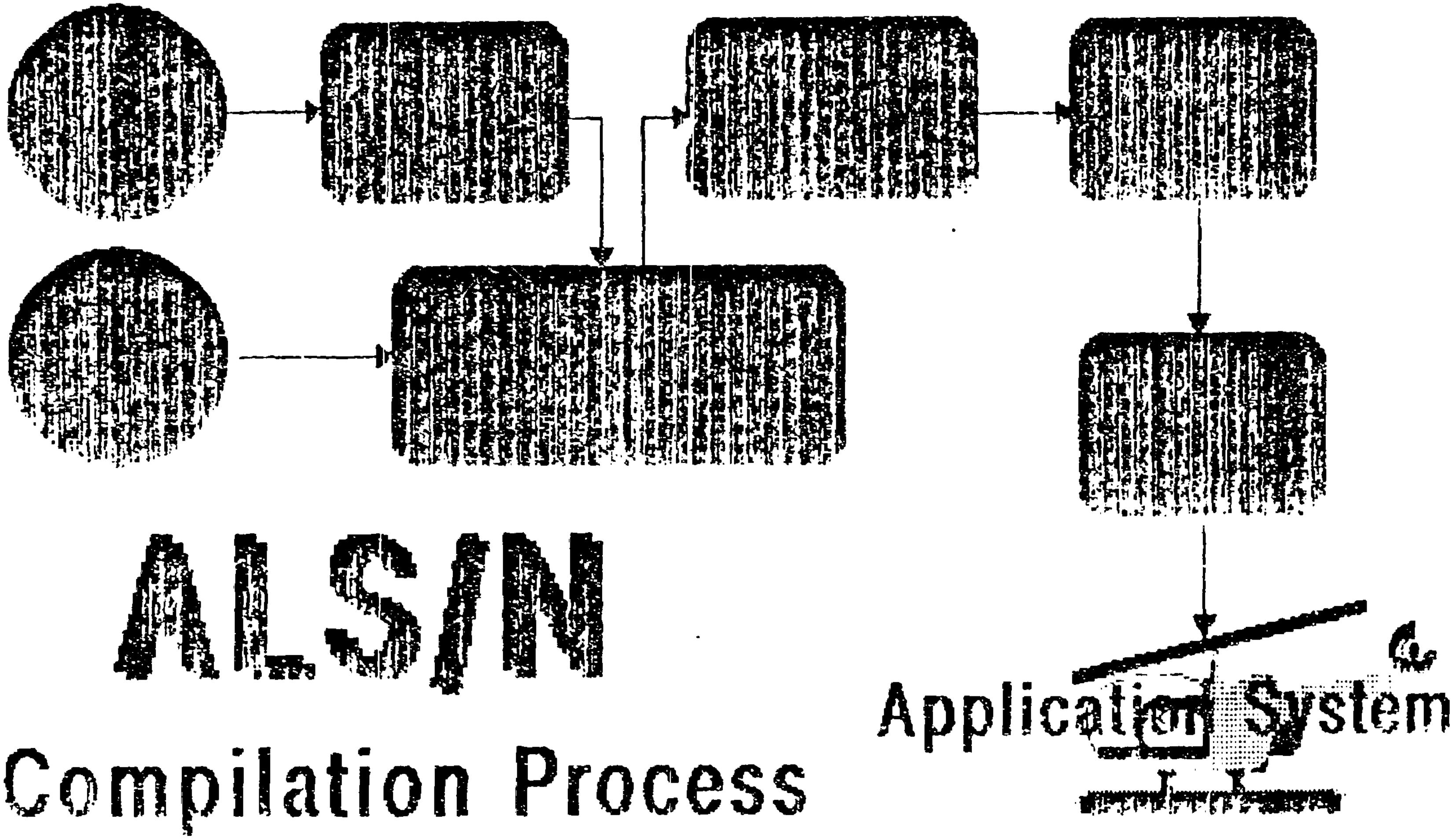
Compilation Process

4/86 NITIADP_25

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA



ALS/N

Compilation Process

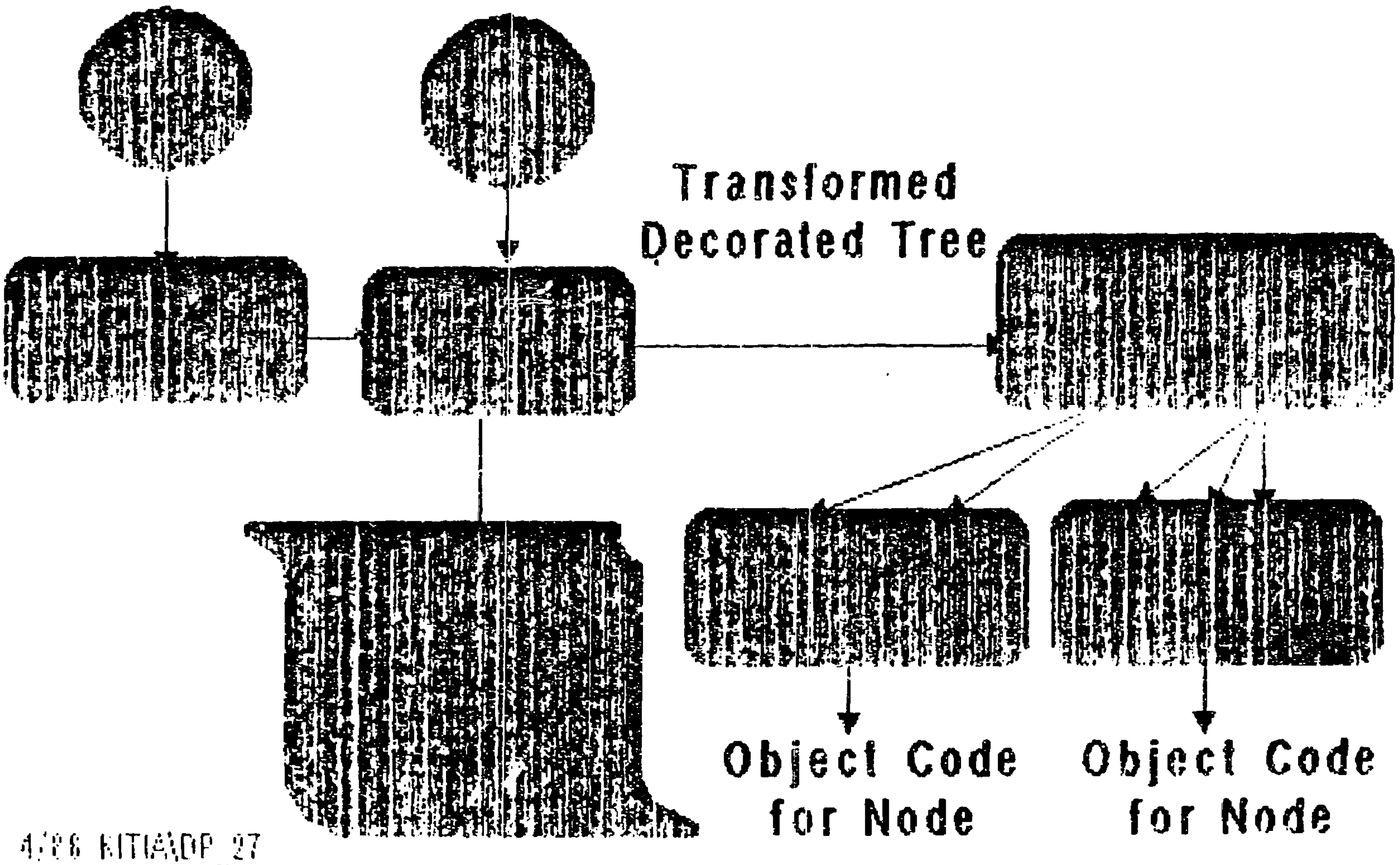
Application System

4/86 KITIADF_26

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

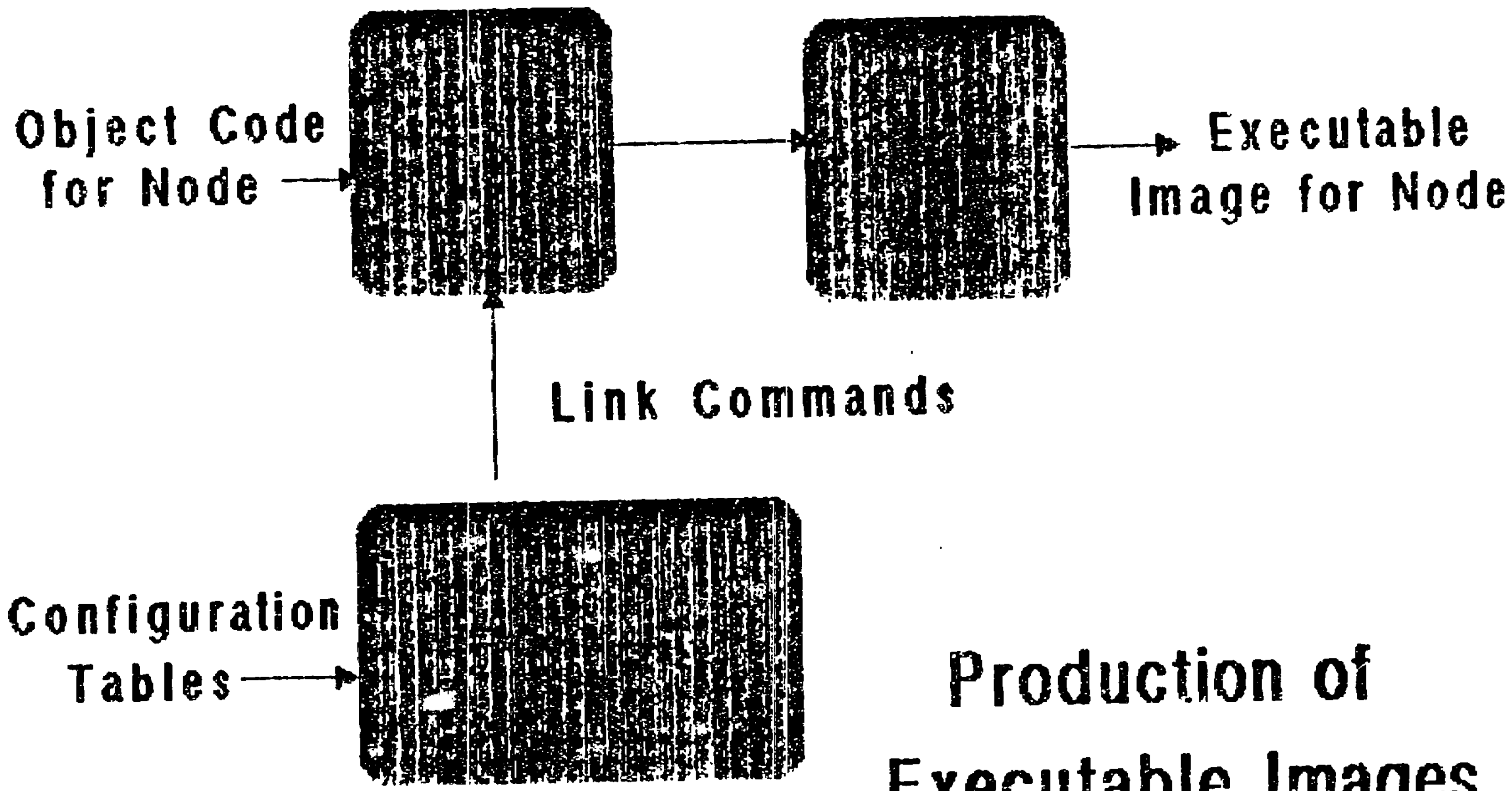


4/86 KITIADP_27

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA



Object Code for Node

Executable Image for Node

Link Commands

Configuration Tables

Production of Executable Images

4/86 KITIA\DP_28

NAVSEA
PMS - 408

ALSN: Multiprogramming Distribution

CONTROL
DATA

SUMMARY:

LOW RISK (non Distr. OS)

IMPROVED PORTABILITY

IMPROVED MAINTAINABILITY

IMPROVED SURVIVABILITY

4/86 KITIA\DP_23

Applying Denotational Semantics to Specifying Kernel Interfaces

R.S. Freedman
Department of Electrical Engineeing and Computer Science
Polytechnic University
Route 110 Farmingdale NY 11735
Freedman @ ADA20

1. Denotational Semantics: Pragmatics

The denotational approach to formal semantics involves specifying abstract mathematical meanings to objects, in such a wag that the meanings of the objects are modelled by the mathematical abstractions. The mathematical entities that are used for this purpose (the denotations) are well-understood classes of sets and functions. The denotational approach is suitable for modelling machine-independent meanings because of its emphasis on mathematical constructs. Consequently, the denotational approach has frequently been used for the formal implementation-independent specification of programming languages, and for deriving rules for proofs of program properties (an axiomatic semantics).

The essential idea in a denotational semantics is to map the syntactical structures (some sets and functions) of a language onto some semantic structures (other sets and functions). This is done so that every legal program in a language can be mapped into its meaning. The approach usually taken is to recursively describe the semantics of a construct in terms of its sub-constructs. The use of the denotational approach is applicable to certain types of sets, called domains, in order to insure convergence in the recursive application of functions. The formal mathematics of this approach was presented by [Scott and Strachey].

There are several notations (or "meta-languages") for specifying a denotational semantics. The most common one, used by [Tennent], [Gordon] and [Stoy] is a variant of Lambda Calculus. This notation, while mathematically precise, is hard to read by many programmers and language implementers. Other notations that have also been proposed include the "Ada-like" notation in the Ada Formal Semantic Definition [INRIA], and the notations developed in the Vienna Definition Method [Bjorner].

Many of these notations have automated facilities that help evaluate and sequence a large number of recursive function calls that stablish the meaning of a construct. For example, [Kini et al] has developed tools for testing the denotational semantic definitions of programmong languages, as long as these languages are defined in AFDL+ (an extension of the INRIA notation). [Mossess] has also developed the Semantics Implementation System based on the notation in [Gordon]. These systems run programs that "execute" the meta-language equations that define the semantics of a construct. In one sense, development of these tools results in an operational semantics of a construct.

Denotational semantics have been used to formally specify programming languages, compilers [Clemmensen], interpreters [Stoy], and databases [Bjorner]. There is also a formal specification of concurrency presented using denotational semantics [Clinger]. Some of the issues involved with specifying kernel facilities based on the denotational approach were first addressed in [Freedman 1982] and [Freedman 1985]. In the following sections, we show what is entailed to develop a denotational semantics for kernel interfaces.

2. Denotational Semantic Domains

The denotational semantics of a kernel interface language consists of the semantics of procedure and function calls, as well as the semantics of expression evaluation. In order to create this denotational semantics, we need to specify the following components:

Syntactic Domains

Syntactic Clauses

Semantic Domains

Semantic Functions

Semantic Clauses

The syntactic domains of a language consists of different syntactic categories that may be assigned meaning. These categories may (recursively) define other categories; to assure convergence, domains are specified. Some examples of syntactic domains are a domain of identifiers, a domain of commands, and a domain of expressions. For CAIS interfaces, these domains consist of identifiers, expressions, commands, and declarations.

The syntactic clauses show how a syntactic category may be described in terms of sub-categories. For example, one clause may specify that all kernel interface commands have the form:

C ::= open(E) | close(E)

where E is in the domain of expressions. The notation for syntactic clauses usually follows the notation for specifying the concrete syntax (phrase structure) of a language. However, since only the meanings of constructs and sub-constructs are emphasised, and not how a construct is formed, this type of syntax is termed the abstract syntax.

The semantic domains consist of well-understood domains that are either given (like the domain Bool = {TRUE, FALSE}) or are constructed from other domains. These domains are the actual "denotations" for our semantics. The most important of these domains are the Environment, the Store, and the Continuation domains. For example, an Environment domain may described by the domain of functions from the domain of identifiers Ide to the domain of denotable values Dv, or

Env = Ide --> Dv

The domain of denotable values must be defined in turns of other domains: the denotable values usually contains the domain of locations. The Environment is changed by the elaboration of definitions. Stores may be described by the domain of functions from the domain of Locations Loc to the domain of Storable Values Sv, or

Stores = Loc --> Sv

Stores are changed by the execution of commandsThe continuation domains may be described by functions from "intermediate results" to "final results." Final program results are usually expressed in terms of the Store domain. For example, since the effect of executing a command is to change the Store, the domain of command continuations is defined by

ComCont = Store --> Store

As another example, since the eff :ct of evaluating expressions is a value and a store (from possible "side-effects"), the domain of expression continuations is

ExpCont = [Dv x Store] --> Store

The above expression may also be written as

ExpCont = [Dv --> Store] --> Store

and also as

ExpCont = Dv --> Store--> Store

This particular form of function notation (the "curried" form) is what makes traditional denotational semantics difficult to read.

The semantic functions are functions that specify the denotation of the syntactic domain constructs in terms of the semantic domain constructs. For example, the semantic function for expressions may be

E: Exp --> Env -->Store --> Dv

This expresses the fact that the semantics of "evaluating an expression" is a value that depends on an environment and a store. Semantic functions are defined for all syntactic domains.

The actual semantics for the constructs that range over all syntactic domains are defined by semantic clauses. A semantic clause is a semantic function definition for a particular syntactic construct. In one sense, the semantic functions form specifications, while the semantic clauses actually "implement" the semantics. For example, the evaluation of the expression "1=1" denotes TRUE, given an arbitrary store s, and an arbitrary environment u:

E [1=1] u s = TRUE

Semantic functions traditionally utilize square brackets around syntactic constructs to increase readability. Other notation for semantic clauses may correspond to more familiar programming language syntax. For example, in the AFDL [INRIA] "Ada-like" notation, the semantic function E for expressions may be represented as

```
function EVAL_EXPRESSION ( T: Syntax_Tree; En: Environment; S: Store)
        return Denotable_Values;
```

The semantic clauses for all expressions would correspond to the function bodies of EVAL_EXPRESSION, for all possible elements of Syntax_Tree. The disadvantage of this notation is its ineconomy: other functions (and the non-Ada like "function type") must be defined to achieve all meanings of the functional notation form for E. For example, E [open (E1,I2,I3,E4)] u is a function, not a value.

7.2. An example of Denotational Semantics for the Specification of Kernel Interfaces

We provide an example of the denotational approach to describe the kernel interfaces of CAIS package Node_Management. This example shows the beginning specification that must be specified for a denotational semantics: the domains Node and Asv, as well as most semantic clauses are left incomplete.

Kernel Facility: package Node_Management

Syntactic Domains

Ide	The domain of identifiers with elements I1,I2, ...
Exp	The domain of expressions with elements E1, E2,...
Com	The domain of commands with elements C1, C2,...
Dec	The domain of declarations with elements D1,D2,...

Syntactic Clauses

```
C ::=  open (E1,I2,I3,E4);
       | close (I1,I2,I3,I4);
       | change_intent (I1,I2,E3);
       | copy_node(I1,I2,I3,I4);
       | copy_tree (I1,I2,I3,I4);
       | rename (I1,I2,I3,E4);
       | link (E1,E2);
       | iterate (I1,I2,I3,E4,E5,E6);
       | get_next (I1,I2);
       | set_current_node (E1,E2);
       | get_current_node (I1);
```

```
E ::=  is_open (I1)
       | kind (I1)
       | primary_name (I1)
       | primary_key (I1)
       | primary_relation (I1)
       | path_key (I1)
       | path_relation (I1)
       | obtainable (I1, I2, E3)
       | more (E1)
       | is_same (E1,E2)

D ::=    I1: node_iterator ;
       | I1: relationship_key_pattern := E1;
       | I1: relation_name_pattern := E1;
```

Semantic Domains

Env The domain of environments with elements u:

Env = Ide --> [Dv + {unbound}]

Dv The domain of denotable values with elements d:

Dv = Loc + Asv + Cc (Exceptions are denotable.)

Loc The domain of locations with elements 1.

Asv The domain of assignable values with elements a.

Store The domain of stores with elements s:

Store = Loc --> [Sv + {unused}]

Sv The domain of storable values with elements v:

Sv = Node + Asv

Node The domain of nodes with elements n.

Cc The domain of command continuations with elements c:

Cc = Store --> Store

Ec The domain of expression continuations with elements k:

Ec = Dv --> Cc

Dc The domain of declaration continuations with elements d:

Dc = Env --> Cc

Semantic functions

Semantics of expressions:

E: Exp --> Env --> Ec --> Cc

Semantics of commands:

C: Com --> Env -->Cc -->Cc

Semantics of declarations:

D: Dec --> Env --> Dc --> Cc

Semantic Clauses (some examples)

Commands

C [open (E1,I2,I3,E4)] u c = (meaning)

Expressions

E [is_open (I1)] u k = (meaning)

Declarations

D [I1: node_iterator] u d = (meaning)

3. Analysis of Denotational Approach

The denotational approach to formal semantics can adequately specify kernel interfaces, provided one interprets these interfaces as defining a language. The complete specification of CAIS semantics for storage management and input/output can also be expressed, although it would be a laborious undertaking, even if aided by automated tools. The major tasks in these areas involves selecting a formal mathematical model for the CAIS data structures and devices. These formal models would then be represented in the notation chosen for the domains. Semantics for process management can also be described in the denotational style, assuming that a formal model of concurrency (like Actor Semantics) is also similarly selected.

The denotational approach is not an alternative method to specifying semantics, rather, it emphasizes a different perspective toward specification. The denotational approach corresponds to a "top-down" solution to the problem of defining a language: the emphasis is on developing mathematical domains and functions to model machine meanings resulting from program execution. The operational approach corresponds to a "bottom-up" solution, whereby the emphasis is on constructing machine operations that will execute programs. An algebraic semantics is also a denotational semantics; in this approach, other specific mathematical constructs are used (more specific then domains) for representing the denotations. As observed above, a denotational specification becomes an operational specification if tools are provided that can "execute" the denotational semantic notation. Both approaches are used to construct rules of program properties to enable an axiomatic semantics.

It is not clear which approach is best for the specification of kernel interfaces. An operational approach would probably easier to understand (but harder to modify or check for consistency or completeness) then a denotational spercification; conversely, a denotational specification is more amenable to a machine independent meaning. This last characteristic is important for achieving interoperability and transportability. On the other hand, the use of denotational semantics for the specification of concurrent computation in Ada has not been as adequately addressed as in some other languages; this implies that for process management, at least many researchers are more comfortable with an operational approach.

REFERENCES

Bjorner,D., Jones, C., Formal Specification and Software Development, Prentice-Hall International Series in Computer Science, 1982.

Clinger, W., Foundations of Actor Semantics, AI-TR-633, MIT Artificial Intelligence Laboratory, May, 1981.

Clemmensen, G., Oest, O., "Formal Specification of an Ada Compiler- A VDM Case Study," Dansk Datamatik Center, 1983-12-31, 1985.

Freedman, R.S., Programming with APSE Software Tools, "Chapter 5: Addendum: Formal Semantics," Petrocelli Books, Inc., Princeton, 1985.

Freedman, R.S., "Specifying KAPSE Interface Semantics," in Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report Volume II (P. Oberndorf, ed.), NOSC TD 552, October, 1982.

Gordon, M., The Denotational Description of Programming Languages: An Introduction, Springer-Verlag, New York 1979.

INRIA, Formal Definition of the Ada Programming Language (Preliminary Version for Public Review), Ada Joint Program Office, November 1980.

Kini, V., Martin, D., Stoughton, A., Tools for Testing the Denotational Semantic Definitions of Programming Languages, ISI/RR-83-112, USC, May, 1983.

Mosses, P., "Compiler Generation Using Denotational Semantics, in Lecture Notes in Computer Science, Vol. 45, Springer-Verlag, New York, 1976.

Scott, D., Strachey, C., "Towards a Mathematical Semantics for Computer Languages," Proceedings of the Symposium on Computers and Automata (ed. J. Fox), Polytechnic Institute of Brooklyn , New York, 1971.

Stoy, J., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, 1979.

Tennent, R., Principles of Programming Languages, Prentice-Hall International, 1981.

APPLYING SEMANTIC DESCRIPTION TECHNIQUES TO THE CAIS

by

Timothy E. Lindquist
Arizona State University
Tempe, Arizona 85287
Lindquis%asu@csnet-relay
(602) 965-2783

Roy S. Freedman
Polytechnic University

Bernard Abrams
Grumman Aircraft Systems

Larry Yelowitz
Ford Aerospace

March 1, 1986

ABSTRACT

Throughout the development of the CAIS, semantics has continued to be an issue. Aside from the benefit to designers, having a formal description of an operating system interface, such as CAIS, is important to CAIS standardization and transportability of software using the CAIS. Although validation systems for kernel interfaces are now being developed, the community has largely ignored kernel interface verification. Constructing proofs of systems that use a kernel clearly depends on a formalism for the kernel. In this paper, various methods of description are analyzed regarding their applicability to kernel interfaces. The methods treated include English narrative, abstract machines, axiomatics, and denotational descriptions. For each method, we show an example from CAIS and analyze the methods applicability to various features.

Keywords. Kernel interfaces, operating systems, verification, axiomatic and denotational semantics.

1. INTRODUCTION

This paper describes and evaluates alternative methods of specifing the semantics of kernel level facilites. Both formal semantic methods and informal methods are examined. The authors have been involved with an effort to develop a common set of services to support APSE (Ada* Programming Support Environment) tools. The methods we describe are exemplified using this common set, called CAIS (Common APSE Interface Set, pronounced as case). CAIS is an operating system interface that supports software development tools.

If CAIS is implemented on a variety of host systems then the effort needed to transport tools will be reduced. In the same manner as for the Ada Language, a validation capability is being developed for the CAIS [E&V 85]. Validation must address the consistency and completeness of CAIS implementations with respect to the specification. In doing preliminary work on developing validation tests, we found the need for a precise specification of the system. Various specification methods have been examined for their applicability to CAIS features. In this paper, we present and compare the applicability of each method.

Although the problem of describing kernel facilities has not received adequate treatement, the benefit of formal description is clear. Any effort to standardize on a low level interface, such graphics or process management, needs a precise specification to be complete enough and unambiguous. As standards arise, we are seeing the development of validation mechanisms to assure consistency among implementations, as mentioned above with CAIS. It has also become clear that formal specifications can be used to direct implementation efforts. Technology is advancing to the point where directed implementations are as efficient as ad hoc implementations.

2. CAIS: A COMMON OPERATING SYSTEM INTERFACE

In order to control the high cost of software that is embedded in military systems, the Department of Defense has developed and is in process of standardizing on a single programming language called Ada. The cost savings will be realized not only from software engineering features of the language, but also from the fact that a single standard will permit the reuse of operational software and software development tools.

When tools are considered, however, a single language is only part of what is needed for transportability. Another requirement is a compatible operating system. APSE tools access environment data and control processes through operating system services. The combination of a standard language and a standard operating system would increase tool transportability.

*Ada is a registered trademark of the U.S. Government Ada Joint Program Office.

Since there are many diverse systems in use today, a single operating system is not feasible, but something almost as good can be achieved. CAIS defines a common interface to the operating system. The interface is a set of Ada packages containing subroutines and data definitions that are used by Ada programs to request system services. Implementations of CAIS may be constructed effectively on a variety of existing operating systems. If the format of the call for services (syntax) is standard and the response to the call (semantics) is the same, then the effect of a standard operating system has been achieved.

CAIS has been designed by a working group of the KIT/KITIA (Kernel APSE Interface Team/Industry and Academic) under sponsorship of the Ada Joint Program Office through Naval Ocean Systems Center [KIT-82]. A Government Standard CAIS specification [CAIS-85] is currently being reviewed, and an effort will soon be underway to address incorporating capabilities deferred from the design, such as distributed environments. Several prototypes and implementations are currently being developed.

3. SPECIFYING KERNEL FACILITIES

3.1 Define Semantics and Syntax

A typical CAIS facility is the OPEN procedure, whose format is shown in Figure 1. The procedure specification gives the procedure name, the keywords, the necessary punctuation, and the parameters. This format is the syntax. The CAIS document accompanies the procedure specification with English narrative telling what happens when the call is executed. The description of what OPEN does is semantics. The usage of words in this context follows the usage in English grammar where syntax is the format of the sentence and semantics is their meaning.

```
procedure OPEN(NODE: in out NODETYPE;
               NAME: in NAMESTRING;
               INTENT: in INTENTION := (1 => READ);
               TIMELIMIT: in DURATION := NODELAY);
```

Figure 1. The OPEN facility's syntax.

Ada provides a well understood notation that completely and unambiguously defines syntax. Ada semantics are conveyed in English text and the Language Reference Manual states that meanings are as defined in Webster's Dictionary. Text benefits from the power and suffers from the ambiguities of a natural language specification. The English description is adequate for most purposes but is often incomplete and ambiguous.

One example is the OPEN statement of Figure 1. Its function is to create an association between an Ada program variable and a CAIS environment node. The internal variable, called a node handle, is used by the program to reference the node in operations. One parameter to OPEN is an array, called INTENT, that conveys intended access. Typical values of Intent are Read, Write, and ExclusiveRead. As an example of incompleteness, the explanation of OPEN does not indicate behavior if the Intent array contains overlaping or contradicting intents. What if both Read and ExclusiveRead are requested? No semantics are defined in this case. Natural language specifications contain implied assumptions about their context. An implied assumption of the open statement may be that a user doesn't care which one of a contradicting intents is chosen. Such a ~~specification~~ specification may be precise enough for a user but not for validating, implementing, or arguing formally about programs using CAIS. Semantics of CAIS are mostly well defined, however, one can anticipate uses requiring a more thorough or formal description.

3.3 Methods of Supplementing a Semantic Definition

The semantics of CAIS is specified in MIL-STD-CAIS by English narrative, with some additional semantics implied by the Ada package specifications. The methods investigated for supplementing CAIS semantics can also be grouped into formal and informal methods. The formal methods are mathematical in nature and include axiomatic, denotational, and abstract machine notations. The informal methods are additional narrative and examples.

4. INFORMAL SEMANTIC SPECIFICATION

The informal methods of supplementing semantics, English narrative and examples, have strong points and weak points. English or other natural language narratives can be verbose, ambiguous, and context dependent. The interpretation of an English sentence depends on the background of the reader. Further, English words have many meanings. My dictionary lists 12 for "be", 33 for "beat", and 15 for "bend". There is, however, no match for the understandability and generality of English. Even texts in theoretical mathematics use more English than mathematical notations to communicate.

Description by examples is done through small programs or parts of programs using CAIS. Test cases from a CAIS test suite would make good examples since these are small programs exercising one CAIS feature. Examples are not general and not concise but are very understandable. When there is a choice of methods of specifing semantics the most precise, concise, and abstract method should be used. Formal mathetical methods , when they are applicable, usually meet these criteria. But there are still many cases where informal methods are needed. The informal methods are supplements and not replacements for formal methods. Examples of the use of informal methods to supplement CAIS semantics follow.

4.1 OPEN Facility

OPEN, as discussed above, establishes a connection between an external file and an internal node handle. Objects in CAIS are managed using a node model. A node can be a file node, a structure (directory) node, or a process node. Figure 2 is an example of the use of OPEN. It is a test case showing how an internal program variable called a node handle is connected to an external node by OPEN. The handle is then used to access the node. The example shows semantics in the sense that it shows how the OPEN procedure is used.

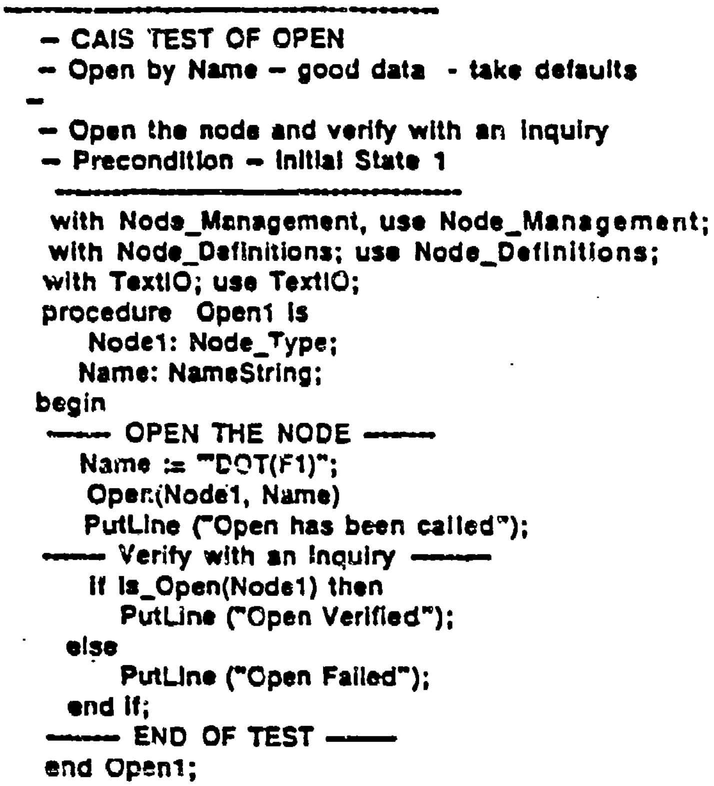
Examples may not show what happens in any of the exceptional cases. Supplementary english narrative can be added showing what happens if, for example, incompatible Intents are presented to the procedure. The Intent array argument to OPEN could be:

(READ, WRITE, CONTROL)

There is nothing to stop a user from specifing an incompatible set of intents. If both WRITE and EXCLUSIVEWRITE are specified there is a conflict. The first Intent lets many users write simultaneously and the second permits only one at a time. This uncertainty can be resolved by additional narrative in the specification such as:

1. In the event of conflict use the most restrictive interpretation; thus, EXCLUSIVEWRITE has precedence over WRITE, or
2. In the event of conflict reject the call with an exception or
3. Any conflict resolution scheme is acceptable.

Any one of the above clarifications is sufficient from the viewpoint of creating validation tests. Which one is chosen is a design issue, however, without specifying one opertion, the validator is forced to make the design decision.



```
------------------------------------
 - CAIS TEST OF OPEN
 - Open by Name - good data  - take defaults
-
 - Open the node and verify with an Inquiry
 - Precondition - Initial State 1
 -----------------------------------
  with Node_Management, use Node_Management;
  with Node_Definitions; use Node_Definitions;
 with TextIO; use TextIO;
 procedure  Open1 Is
     Node1: Node_Type;
    Name: NameString;
begin
 ---- OPEN THE NODE ----
    Name := "DOT(F1)";
     Open(Node1, Name)
     PutLine ("Open has been called");
 ---- Verify with an Inquiry ----
     If Is_Open(Node1) then
        PutLine ("Open Verified");
   else
        PutLine ("Open Failed");
   end if;
 ---- END OF TEST ----
 end Open1;
```

Figure 2 Example of Open Procedure

4.2 The CAIS Node Model

Nodes, node handles, and path names are part of the node model. CAIS manages files, directories, devices, and processes by representing them as nodes in a network. Nodes are related to each other by relationships. Relationships are uniquely specified by a relation name and a relationship key, and a relationship may be either primary or secondary. Primary relationships are constrained to maintain a hierarchical structure of nodes. A typical network is shown in Figure 3.

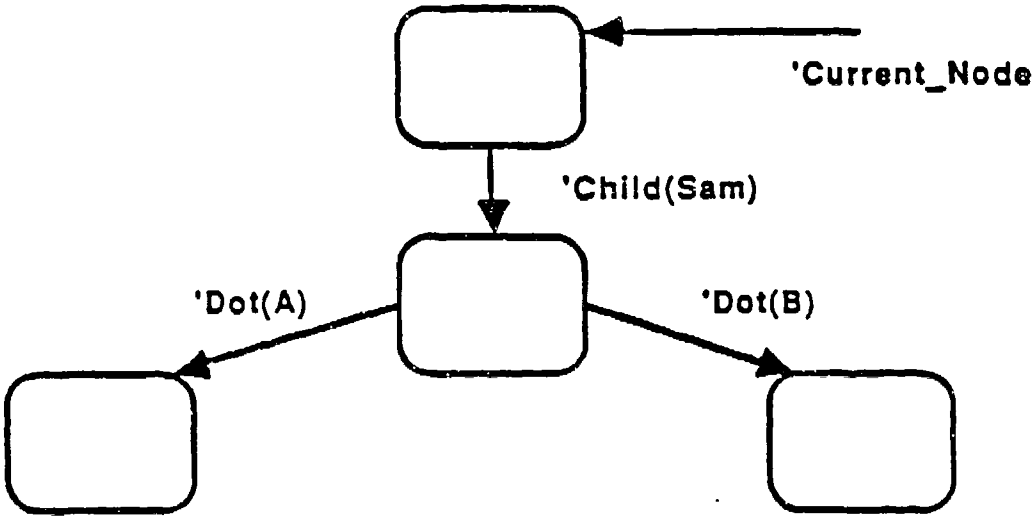


Figure 3. The Node ModeL

An object, such as a file, is found by following the path of relationships from a known node to an object node. A path is specified by a path name made by concatenating all the element names along the path.

Another CAIS function is PRIMARY_NAME. The input to the function is a node handle, and the function returns the name of the primary path to the node. The Priminary Name function returns the path name. For example the path from node Joe to node Sam is: 'Current_Node'Child(Sam). Since Current_Node is the default relation, the path can also be expressed as: 'Child(Sam). Child is the name of a relation. Sam is a particular instance of the relation. Since Sam is the only instance, the relation key "Sam" can be omitted and the path expressed as: 'Child . The relationship 'DOT(A) uses the default relation name (DOT) that can be expressed in two ways, Dot or (.). Two of many ways of expressing the path between the current node and "A" are:

'Child(Sam)'Dot(A) or 'Child(Sam).A

It is clear that the semantics are ambiguous. There are many different strings that could be returned by the function. If the intended meaning of the designers was that any valid name string is acceptable, this could be stated in one sentence. However, allowing any string to be returned may promote implementations that hinder transportability. One way to supplement the semantics is with the following paragraph:

> The full path name up to the Current_Node shall be returned. Relationship keys shall be spelled out even when they are unique. The dot relation shall be in the long form.
>
> An example of a correct name using the network of Figure 3 is:
>
> 'Child(Sam)'Dot(A)

4.3 Conclusion on Informal Methods

Using English narrative and examples to supplement CAIS semantics has strengths and weakensses. Any specific ambiguity in the CAIS specification can be corrected by a combination of narrative and/or examples without requiring a formal description. Examples and narrative have the advantage of being easy to construct and comprehensible. A large portion of CAIS can be specified in a short description. The short description provides a quick introduction to which details can be added. The primary disadvantage is the difficulty in obtaining completeness in narrative specifications. Narratives cannot be used for formal arguments of correctness or arguments of interface characteristics. Descriptions using these techniques can best be viewed as a step in the process of developing more complete and formal descriptions. The most useful form of narrative is one that is developed in conjunction with or based on a formal description. Doing so reduces the tendency toward incompleteness or ambiguity.

5. ABSTRACT MACHINE DESCRIPTION

A report from a preliminary study of validation in an APSE [KAF-82] indicates that specifying the semantics of an interface such as CAIS requires more than a description of the syntax and functionality of its routines. Interactions that exist at the interface must be specified. Interactions may include routines that operate on a common data structure, routines that rely on data produced by a tool or routines depending on the Ada runtime environment. Furthermore, any pragmatic limits which apply to implementations must also be specified. These might include the length of identifier strings, field sizes, maximum number of processes, or the maximum number of times that an interface routine may be called.

Lindquist [LIN-84], describes an Ada-based Abstract Machine approach to describing CAIS. Using this approach functionality is operationally described in the form of Abstract Machine Programs. The programs are written in Ada. One is written for each CAIS routine to describe what that routine does. If there existed an executor for the programs (the Abstract Machine) then an operational definition of CAIS would exist. In later papers, [LIN-85a, SRI-85], the technique is demonstrated using the CAIS process model and applied to the problem of generating a validation mechanism for CAIS implementations. As depicted in Figure 4, an Abstract Machine consists of three components:

1. A processor,
2. A storage facility, and
3. An instruction set.

The processor is able to recognize and execute instructions from a predefined set. Each instruction has an action that the processor carries out in some data context. One component of the processor, called the environment pointer, indicates the data context in which an instruction is to be executed. Another component, called the instruction pointer, sequences processor execution through the instructions of the program.

The storage of the processor is memory for both data and programs. Data storage constitutes the environment used by the processor to execute instructions. The final component is the instruction set.

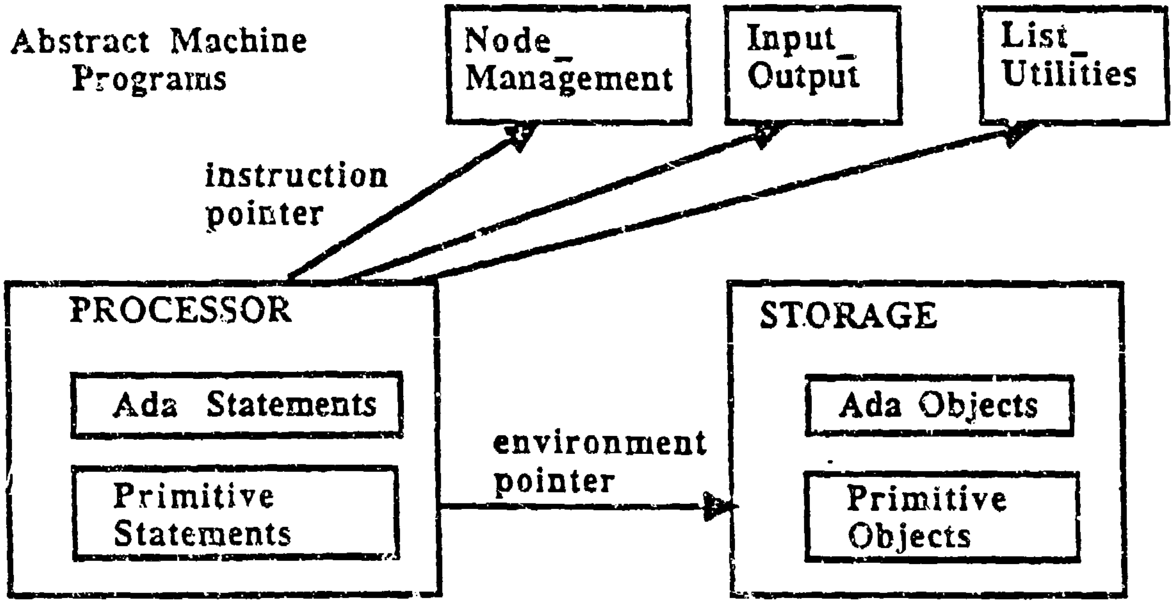


Figure 5.1. CAIS Abstract Machine

Figure 4. CAIS Abstract Machine.

Instructions are taken from the Ada language and are augmented needed primitive operations. The meanings of these primitive operations are left to the description of the Abstract Machine. Additional operations can be viewed as extending the instruction set of the Abstract Machine to include operations beyond the scope of Ada.

5.1 Ada Abstract Machines to Describe CAIS

Although other Abstract Machines could be used, this section presents one that is Ada-based. Several aspects of Ada make it a desirable choice for this description. One is the richness of the Ada control constructs and typing facilities. Further support for using Ada lies in the observation that any language used as a semantic description tool must have a well-defined semantics of its own. Although Ada has not been totally specified using a formal technique, the language's controlled definition provides an adequate basis for the Abstract Machine. The most compelling reason for using Ada is compatibility with the uses of the CAIS specification. CAIS implementors and users are familiar with Ada, thus making an Ada-based Abstract Machine more comprehensible and useful.

5.1.1 Node Management and List Utilities.

CAIS defines a set of list manipulation facilities that may be used in conjunction with the CAIS. Lists may be either named or unnamed. Named lists are those in which each element in the list has a unique name. The package includes routines for constructing generalized lists containing string, interger, list, and floating point elements. Routines to add, remove, and examine elements of a list are provided. An Ada-based Abstract Machine description of List Utiulities follows the same approach as an Ada implementation. Figure 5 demonstratres the linking structure our definition uses for the example named list:

(APPLE => "GREEN", GRAPE => (RED => "SEEDLESS"))

List manipulation routines are constructed in Ada using this representation. A criticism of Abstract Machine descriptions is that the code itself specifies an implementation technique. Independent of the machinery selected, instructions to carry-out an operation must indicate an implementation technique. The meanings of the routines are not, however, derived from the code, but instead by the effect of executing the code on the Abstract Machine.

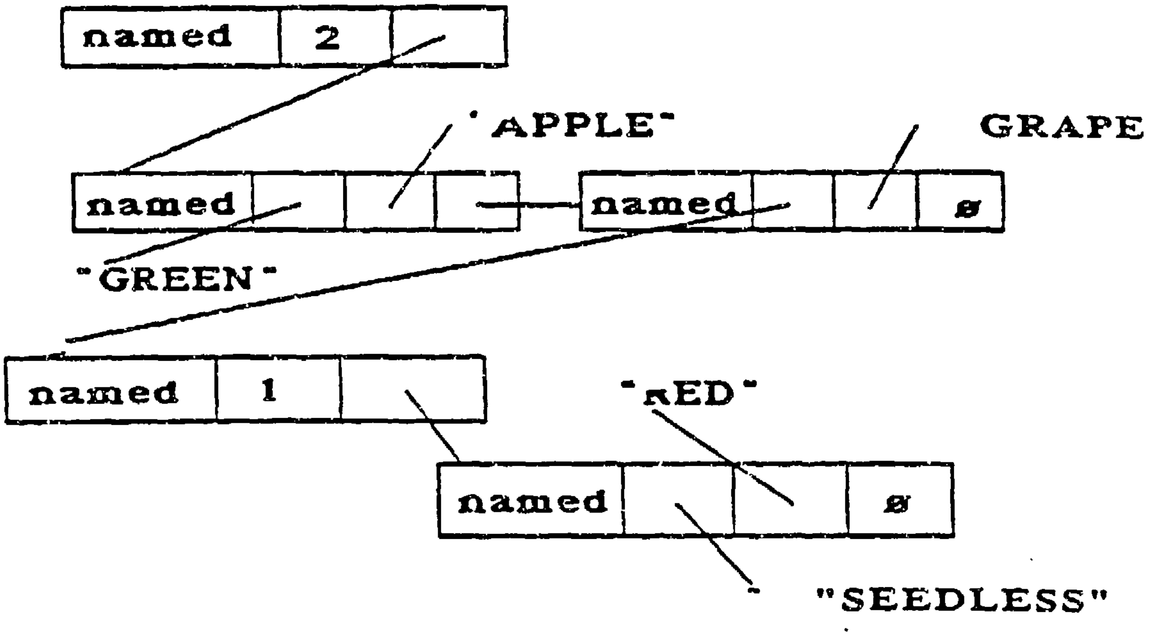


Figure 5. A sample list implementation.

The CAIS Node Management package includes facilities for manipulating nodes, which represent entities of the CAIS environment. Nodes may exist for processes, files, devices, queues, and node structures. Nodes may be related to one another using either restricted or unrestricted relationships. The restricted form of relationships, called primary relationships, require that each node have exactly one parent (except a single root node). The unrestricted form allows more general (cyclic) relationships to exist among nodes. CAIS Node Management provides routines for manipulating nodes, relationships among nodes, and attributes (either node or relationship attributes). Node Management also includes access mechanisms, which control the operations that a process may perform on a node.

The Abstract Machine description of Node Management relies heavily on data mechanisms of Ada. Included in the description are substantial uses of dynamic structures to represent nodes, to store relationships between nodes, and to store lists and attributes. Access types, constrained record types and exception handling mechanisms are all used in the description.

Exception handling facilities are used throughout the CAIS to return status information to the tool calling CAIS. Using Ada exception mechanisms in the Abstract Machine provides an excellent definition of status returns for CAIS. With any other formal semantic description (axiomatic or denotational) a reasonable overhead equal to describing Ada language exceptions is incurred.

One use of Ada exceptions within Abstract Machines illustrates the problem of over specifying semantics. For instance, suppose the CAIS specification indicates an incomplete order for generating status exceptions to allow for flexible implementations. Thus, when a CAIS routine is called with arguments that would produce multiple status exceptions, the specification does not impose a complete order for checking. An Abstract Machine description does, however, fully specify the order of status checking.

5.1.2 Process Control.

The Process Control section of CAIS provides routines to create and manage the execution of Ada programs. Facilities are included for different forms of invoking processes, awaiting completion, and manipulating built-in process attributes.

The Abstract Machine description relies on Ada's tasking facilities to describe asynchronous processes in the CAIS environment. For example, in the Abstract Machine description [SRI-85], a process node is represented as a dynamically created (allocated) record object. Components of the object contain instances of task types which provide the parallelism and synchronization needed for spawing and awaiting processes. A user's process structure is built dynamically and is a tree of tasks. Each process includes tasks for synchronization and for representing the Ada program. An example of two CAIS processes is shown in Figure 6. Process_node_1 has spawned Process_node_2, and the spawned_process task is used to synchronize among processes. Again, the use of Ada's tasking facilities in the Abstract Machine description alleviate the need to formally redefine asynchronous facilities in some other descriptive technique. Both axiomatic and denotational approaches have a cumbersome time accommodating concurrency. Tasking is well understood by the users of a CAIS specification, which eases comprehension. However, we note that a formal specification of Ada tasking does not yet exist.

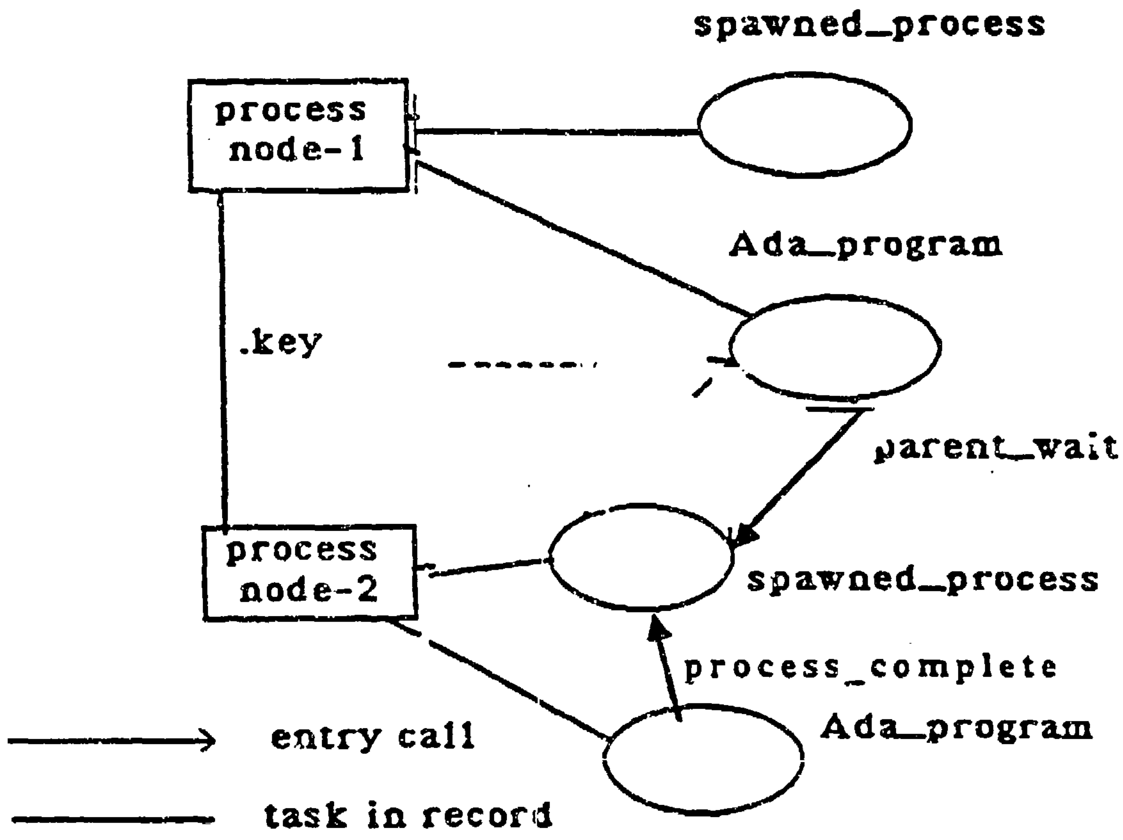


Figure 6. Sample CAIS process structure.

5.1.3 Input and Output.

Routines for manipulating file nodes of various types are included in the Input/Output section of CAIS. Further support is provided for common types of terminals and magnetic tapes. To construct an Abstract Machine description of this section of CAIS, the Abstract Program must create software devices that appear to the CAIS just as actual devices would appear. While it is possible to define a majority of the input/output facilities using an Ada-based Abstract Machine, some routines do not lend to formal specifications using any technique. Facilities to require the operator to physically mount or dismount tapes from a drive exemplify those difficult to define formally. Although one could formally define routines requesting these services, the need to formally define such facilities can be argued.

5.3 ANALYSIS OF ABSTRACT MACHINES

An Ada-based Abstract Machine description of CAIS provides some distinct advantages in the progression to a more formal specification of CAIS. Some of theses advantages would be lost if the Abstract Machine description were to be formulated in a language other than Ada. For example, an Ada-based description can be constructed quickly. If another language were used, then the problems of translating the meanings of asynchronous activity and exception handling into the notation of that language would need to be overcome. Additionally, an Abstract Machine description in another language would not be as comprehensible to the Ada commmunity as is a specification in Ada. To be a complete formal specification of CAIS, an Ada-based Abstract Machine description must be accompanied by an appropriate formal specification of the machine instruction set. Inventing and defining appropriate instructions to augment Ada could be done to deal with drawbacks such as over specification.

6. AXIOMATIC DESCRIPTION

One of the ubiquitous comments received from the public review of CAIS 1.1 is the need for a semantics. There are a variety of methods for presenting a formal semantics, and this section treats the axiomatic approach. There is no escaping the fact that some degree of mathematical maturity is required to comprehend any formal semantics. It is our feeling, however, that axiomatic semantics is the most comprehensible to the largest set of serious CAIS readers.

Axiomatics was first presented by Hoare [HOA-69], and has been applied to various languages; the most notable of which is PASCAL [HOA-73]. London [LON-78] has applied the method to EUCLID, which is especially interesting since the langauge was designed with the goal of simplifying program proofs. A large portion of this presentation is based on Yelowitz [YEL-84].

In a mathematical sense, a theory is defined by applying the Axiomatic method to a programming langauge. The theory consists of a language for expressing theorems, a set of axioms and rules of inference. A theorem of the theory is a program together with its input and output specifications. Minimally, it is required that all theorems of the system be programs which match their specifications; that is, the system must be sound. Axioms and rules of inference are defined to determine whether or not a program and its specifications form a theorem. If a program and its specifications are derivable from the axioms and rules, then they constitute a theorem.

By derivable, we mean that there exists a proof of the theorem in the system. A proof is a sequence of statements in the theory that begins with an axiom and ends with the theorem. Each statement in the sequence is either an axiom or it is a statement that can be written by applying a rule of inference to statements preceeding it in the proof.

Syntactically, the theorems of the system take the form:

$$\vdash P \{ S \} Q.$$

Where S is a statement or set of statements of the programming language and P and Q are predicates (assertions) over the variables used in S. Our statements are Ada langauge statements augmented by calls to CAIS interfaces. The turnstile, |--, indicates that P{S}Q is a theorem of the system. Intuitively, P{Q}S can be interpreted to mean, if P is true before execution of S, then Q will be true after execution provided S halts.

An axiomatic semantic description of CAIS can be formulated in conjunction with that of the Ada language. Assuming that such a definition of the language already exists, we outline here how it may be augmented to accommodate CAIS. CAIS interfaces may be treated in the same manner as other procedures or functions invoked by an Ada program. Input and output predicates may be constructed to define what the procedure does. The free variables of the predicates are the parameters and nonlocals referenced by the procedure. A rule for the CALL statement defines how the input and output assertions are used to prove procedure calls. Although input and output assertions could be defined in this manner, we choose to represent the meaning of CAIS facilities with Axioms (schemes) to more accurately reflect the relationship between CAIS facilities and the language.

We now present the background needed for the Axiom scheme for the Node Model routine COPY_NODE. To do so requires formalization of notions such as types of nodes, contents of nodes, attributes of nodes, and relationships among nodes.

6.1 The CAIS Node Environment

The node environment can be described as a directed graph in which arcs are labeled and may possess attributes. We define NODES to be the set of nodes in an APSE.

The set ARCS includes all directed edges in the graph. Thus:

$$\text{ARCS} \ \varepsilon \ \text{NODES} \ \text{X} \ \text{NODES}$$

If the pair (n_1,n_2) ε ARCS then there is a directed edge from n_1 to n_2. We refer to an element of ARCS with the shorthand a_i Labels formalize the relationships that ARCS represent. LABELS is a set of the

(relation_name, relationship_key)

pairs associated with each arc in CAIS. The function LABEL names each arc with the appropriate pair as:

$$\text{LABEL} : \text{ARCS} \rightarrow \text{LABELS}$$

A pathname is a sequence of labels. Thus all valid pathnames are in the Kleene star of

LABELS (LABELS*).

OUTARCS is a function providing for each node, a set of all arcs emanating from the node.

$$\text{OUTARCS} : \text{NODES} \to 2^{\text{ARCS}}$$

That is, an edge is in the set of out arcs of a node, n, when it emanates from n.

$$a \in \text{OUTARCS}(n) \quad \text{iff} \quad \exists\, n_1 \in \text{NODES and } a = (n, n_1) \in \text{ARCS}$$

Similarily we define INARCS to be the set of all edges emanating to a node.

$$\text{INARCS:} \quad \text{NODE} \to 2^{\text{ARCS}}$$

The predicate ISPRIMARY partitions the set of arcs into primary and secondary relationships. CAIS requires that all primary relationships maintain the hierarchical structure of nodes. We describe this requirement using the following:

$$\text{ISPRIMARY} : \text{ARCS} \to \{\text{true, false}\}$$

Any node (except the system root) must have exactly one primary relationship emanating to it. This CAIS requirement can be expressed as:

$$\forall\ n \in \text{NODES} \mid n \neq \text{SYSTEMROOT and } \forall\, a,b \in \text{INARC}(n)$$
$$\text{SINK}(a) = \text{SINK}(b) \implies \text{not ISPRIMARY}(a) \text{ or not ISPRIMARY}(b)$$

Where SINK is the node an arc emanates to: SINK: ARC -> NODE

CAIS specifies that distinct arcs emanating from a node must have distinct labels. To describe this property, we have the following predicate:

$$\forall\ n \in \text{NODES},\ \forall\ a_1, a_2 \in \text{ARCS}$$

$$a_1, a_2 \in \text{OUTARCS}(n) \text{ and } a_1 \neq a_2 \implies \text{LABEL}(a_1) \neq \text{LABEL}(a_2)$$

For notational convenience, we define the following:

$\forall\ (x,y) \in \text{ARCS},\ R \in \text{LABELS}$

P(x,R,y) denotes ISPRIMARY((x,y)) and LABEL((x,y)) = R

P(x,R,y) means there is a primary relationship from x to y labeled R, and

S(x,R,y) denotes not ISPRIMARY((x,y)) and LABEL((x,y)) = R

S(x,R,y) means there is a secondary relationship form x to y labeled R.

There is a partitioning of the set of nodes into four disjoint subsets:

PROCESS_NODES,
STRUCTURAL_NODES,
FILE_NODES,
DEVICE_NODES, and
QUEUE_NODES.

These subsets, which represent the different types of CAIS nodes, allow the axiomatic description to distinguish characteristics particular to different ypes.

6.2 SEMANTICS OF COPY NODE

This interface is used to make a copy of a file or structural node having no primary relationships emanating from it. Secondary relationships emanating from the node are copied, as appropriate. The syntax of one overload of the routine is:

```
procedure COPY_NODE (FROM,TO_BASE:  in NODE_TYPE;
                                    TO_KEY:       in RELATIONSHIP_KEY;
                                    TO_RELATION   in RELATION_NAME :=
                                     DEFAULT_RELATION);
```

Our goal is a predicate transformer for each interface of the CAIS. Since the transformers can be quite extensive, we present one by its parts. A shorthand notation is also used to avoid complexity. The predicate transformer for a NAME_ERROR is:

$$(\text{NOT } (\text{RLN,KEY}) \ \varepsilon \ \text{LABELS}) \text{ or}$$

$$(\exists \ n \ \varepsilon \ \text{NODES} \mid (\text{BASE},n) \ \varepsilon \ \text{ARCS and LABEL}((\text{BASE},n)) = (\text{RLN,KEY}))$$

$$\{ \text{ COPY\_NODE (FROM, BASE, KEY, RLN) } \}$$

$$\text{NAME\_ERROR}$$

The meaning of this transformer is: if prior to executing the call to COPY_NODE the relation name, relationship key pair are either illegal or the node to be created already exists then, if execution of COPY_NODE completes, the predicate NAME_ERROR will be satisfied. To simplify and continue the example, we present an abbreviated form of the transformers for USE_ERROR, STATUS_ERROR and the functionality of COPY_NODE

USE_ERROR is generated according to the following predicates. First, USE_ERROR is raised when there is a primary arc emanating from the source of copying.

$$\exists \ n \ \varepsilon \ \text{NODES} \mid (\text{FROM}, n) \ \varepsilon \ \text{ARCS and PRIMARY}((\text{FROM},n))$$

Next, when the node to be copied (FROM) isn't either a FILE_NODE, or a STRUCTURAL_NODE, USE_ERROR is generated:

$$\text{not FROM} \; \varepsilon \; \text{FILE\_NODES} \quad \text{and} \quad \text{not FROM} \; \varepsilon \; \text{STRUCTURAL\_NODES}$$

The status of a node is defined by the function NODE_STATUS as:

$$\text{NODE\_STATUS} : \text{NODES} \rightarrow \{\text{OPENED, CLOSED, UNOBTAINABLE, NONEXISTENT}\}$$

With this we can define the predicate transformer producing a STAUS_ERROR as:

$$\text{NODE\_STATUS(FROM)} \neq \text{OPENED or NODE\_STATUS(BASE)} \neq \text{OPENED}$$

Normal Action. With these definitions for exceptional conditions, we can define the predicate transformer for a call to COPY_NODE in which copying takes place. The exceptional status conditions given above can all be placed into a single predicate transformer. To do so, the precondition for each precludes the others, as does the corresponding postcondition. Each unit of the predicate transformer corresponds to a separate action. Below is the transformer describing normal operation of COPY_NODE. The precondition is abbreviated as not STATUS_EXCEPTION to indicate that no status returns occur. In that instance, and only in that instance, the copying takes place.

$$\begin{gathered}
\text{not STATUS\_EXCEPTION } \{\text{COPY\_NODE(FROM, BASE, RLN, KEY)}\} \\
\exists \, n \; \varepsilon \; \text{NODES} \mid \text{(BASE, N)} \varepsilon \text{ ARCS and P(BASE,(RLN,KEY),n)} \\
\text{and LABEL((BASE,n))} = \text{(RLN,KEY)} \quad (0) \\
\text{and CONTENTS(n)} = \text{CONTENTS(FROM)} \quad (1) \\
\text{and (ATTRIBUTES(n)} = \text{ATTRIBUTES(FROM)} \quad (2) \\
\text{and KIND(n)} = \text{KIND(FROM)} \; (3) \\
\text{and } \exists \, a \; \varepsilon \text{ ARCS} \mid a = \text{(n,BASE) and LABEL(a)} = \text{(PARENT)} \\
\text{and S(n,(PARENT),BASE} \quad (4) \\
\text{and } \forall \, a \; \varepsilon \text{ ARCS} \mid a = \text{(FROM, FROM)} \; \exists \, b \; \varepsilon \text{ ARCS} \mid \\
b = \text{(n,n) and LABEL(a)} = \text{LABEL(b)} \quad (5) \\
\text{and } \forall \, a \; \varepsilon \text{ ARCS} \mid a \neq \text{(FROM,FROM) and LABEL(a)} \neq \text{(PARENT)} \\
\exists \, b \; \varepsilon \text{ ARCS} \mid b = \text{(n,SINK(a)) and LABEL(b)} = \text{LABEL(a)} \quad (6)
\end{gathered}$$

The postcondition for normal operation is lengthy, so its components are explained by line number. Line (0) indicates that a new node, n, has been created with a primary relationship emanating from BASE to the node. The relation and key are as specified through arguments. Note, however, that the CAIS indicates that the key may not be the argument. If a '#' appears as the key or appended to the key, then CAIS returns a unique key. This could be expressed axiomatically by adding additional conjuncts to both the pre and post assertions.

Lines (1), (2), and (3) indicate that the contents, attributes, and kind of the copied node match the original. Lines (4), (5), and (6) describe the newly created relationships emanating from the copy. Line (4) indicates that the secondary relationship, parent, for the copied node is set to BASE. CAIS indicates that any secondary relationships that emanate from the node to be copied must exist in the copied node as relations emanating back to the copied node. Line (5) defines this situation. Note that it is not necessary to specify only secondary relationships in the predicate since there are no primary relationships emanating from a node to itself. Line (6) indicates that there exists a secondary relationship in the copied node for all others of the from node. Thus, for all arcs from FROM, which don't point to FROM and which aren't parent relationships, there is a corresponding arc from the copied node with the same destination and label.

6.3 Analysis of Axiomatic Semantics

Axiomatic descriptions that rely on first order predicate calculus, which we have assumed here, can be characterized as removing all temporal information from the description. Having no order specified alleviates the problem of over specification that was found with Abstract Machines. Since time is not specified, one is tempted to state that some forms of status returns from kernel interfaces can't be specified. For instance, suppose the kernel indicates that "when conditions for status A and status B are both satisfied, that A is to be signaled". This can, however, be described axiomatically with predicates indicating that B is raised only when the conditions causing it exist and those causing A don't.

There are, however, two problems arising from the lack of temporal information. First, aliases may exist. In CAIS, two names within a CAIS implementation may refer to the same object. For example, suppose a single object is used as the argument to two or more in/out parameters for an interface. To answer the question: which value produced for the parameters will be given to the argument, requires temporal information about the implementation. Second, asynchronous and parallel computations require greater descriptive capability. The inability to specify time dependencies also implicates the inability to specify time independencies. The CAIS process model provides interfaces for concurrently executing processes, as well as for communication and synchronization among processes.

Exception handling causes no problem to axiomatic descriptions providing that the routine signaling the exception also handles it. In the CAIS this is rarely the situation. Exceptions are used to return status information. Although an axiomatic description can be generated to indicate that a status exception has been raised, the action performed to handle the exception is cumbersome to describe. Thus although we can describe the CAIS, we can't describe the meaning of a program that uses CAIS facilities. Binding a raised exception to a handler in Ada depends on the execution flow through the program. The procedure call history is needed when nested procedure calls are made. Ada's rule for binding exceptions requires that the exception be propagated outward to all calling procedures until one containing a handler is found. The program execution path needed for this binding is not available from static analysis.

7. DENOTATIONAL DESCRIPTION

7.1 Denotational Semantics: Pragmatics

The denotational approach to formal semantics involves specifying abstract mathematical meanings to objects, in such a way that the meanings of the objects are modelled by the mathematical abstractions. The mathematical entities that are used for this purpose (the denotations) are well-understood classes of sets and functions. The denotational approach is suitable for modelling machine-independent meanings because of its emphasis on mathematical constructs. Consequently, the denotational approach has frequently been used for the formal implementation-independent specification of programming languages, and for deriving rules for proofs of program properties (an axiomatic semantics).

The essential idea in a denotational semantics is to map the syntactical structures (some sets and functions) of a language onto some semantic structures (other sets and functions). This is done so that every legal program in a language can be mapped into its meaning. The approach usually taken is to recursively describe the semantics of a construct in terms of its sub-constructs. The use of the denotational approach is applicable to certain types of sets, called domains, in order to insure convergence in the recursive application of functions. The formal mathematics of this approach was presented by [Scott and Strachey].

There are several notations (or "meta-languages") for specifying a denotational semantics. The most common one, used by [Tennent], [Gordon] and [Stoy] is a variant of Lambda Calculus. This notation, while mathematically precise, is hard to read by many programmers and language implementers. Other notations that have also been proposed include the "Ada-like" notation in the Ada Formal Semantic Definition [INRIA], and the notations developed in the Vienna Definition Method [Bjorner].

Many of these notations have automated facilities that help evaluate and sequence a large number of recursive function calls that stablish the meaning of a construct. For example, [Kini et al] has developed tools for testing the denotational semantic definitions of programmong languages, as long as these languages are defined in AFDL+ (an extension of the INRIA notation). [Mossess] has also developed the Semantics Implementation System based on the notation in [Gordon]. These systems run programs that "execute" the meta-language equations that define the semantics of a construct. In one sense, development of these tools results in an operational semantics of a construct.

Denotational semantics have been used to formally specify programming languages, compilers [Clemmensen], interpreters [Stoy], and databases [Bjorner]. There is also a formal specification of concurrency presented using denotational semantics [Clinger]. Some of the issues involved with specifying kernel facilities based on the denotational approach were first addressed in [Freedman 1982] and [Freedman 1985]. In the following sections, we show what is entailed to develop a denotational semantics for kernel interfaces.

7.2 Denotational Semantic Domains

The denotational semantics of a kernel interface language consists of the semantics of procedure and function calls, as well as the semantics of expression evaluation. In order to create this denotational semantics, we need to specify the following components:

Syntactic Domains
Syntactic Clauses
Semantic Domains
Semantic Functions
Semantic Clauses

The syntactic domains of a language consists of different syntactic categories that may be assigned meaning. These categories may (recursively) define other categories; to assure convergence, domains are specified. Some examples of syntactic domains are a domain of identifiers, a domain of commands, and a domain of expressions. For CAIS interfaces, these domains consist of identifiers, expressions, commands, and declarations.

The syntactic clauses show how a syntactic category may be described in terms of sub-categories. For example, one clause may specify that all kernel interface commands have the form:

C ::= open(E) | close(E)

where E is in the domain of expressions. The notation for syntactic clauses usually follows the notation for specifying the concrete syntax (phrase structure) of a language. However, since only the meanings of constructs and sub-constructs are emphasised, and not how a construct is formed, this type of syntax is termed the abstract syntax.

The semantic domains consist of well-understood domains that are either given (like the domain Bool = {TRUE, FALSE}) or are constructed from other domains. These domains are the actual "denotations" for our semantics. The most important of these domains are the Environment, the Store, and the Continuation domains. For example, an Environment domain may described by the domain of functions from the domain of identifiers Ide to the domain of denotable values Dv, or

Env = Ide --> Dv

The domain of denotable values must be defined in turns of other domains: the denotable values usually contains the domain of locations. The Environment is changed by the elaboration of definitions. Stores may be described by the domain of functions from the domain of Locations Loc to the domain of Storable Values Sv, or

Stores = Loc --> Sv

Stores are changed by the execution of commandsThe continuation domains may be described by functions from "intermediate results" to "final results." Final program results are usually expressed in terms of the Store domain. For example, since the effect of executing a command is to change the Store, the domain of command continuations is defined by

ComCont = Store --> Store

As another example, since the effect of evaluating expressions is a value and a store (from possible "side-effects"), the domain of expression continuations is

ExpCont = [Dv x Store] --> Store

The above expression may also be written as

ExpCont = [Dv --> Store] --> Store

and also as

ExpCont = Dv --> Store--> Store

This particular form of function notation (the "curried" form) is what makes traditional denotational semantics difficult to read.

The semantic functions are functions that specify the denotation of the syntactic domain constructs in terms of the semantic domain constructs. For example, the semantic function for expressions may be

E: Exp --> Env -->Store --> Dv

This expresses the fact that the semantics of "evaluating an expression" is a value that depends on an environment and a store. Semantic functions are defined for all syntactic domains.

The actual semantics for the constructs that range over all syntactic domains are defined by semantic clauses. A semantic clause is a semantic function definition for a particular syntactic construct. In one sense, the semantic functions form specifications, while the semantic clauses actually "implement" the semantics. For example, the evaluation of the expression "1=1" denotes TRUE, given an arbitrary store s, and an arbitrary environment u:

E [1=1] u s = TRUE

Semantic functions traditionally utilize square brackets around syntactic constructs to increase readability. Other notation for semantic clauses may correspond to more familiar programming language syntax. For example, in the AFDL [INRIA] "Ada-like" notation, the semantic function E for expressions may be represented as

function EVAL_EXPRESSION (T: Syntax_Tree; En: Environment; S: Store) return Denotable_Values;

The semantic clauses for all expressions would correspond to the function bodies of EVAL_EXPRESSION, for all possible elements of Syntax_Tree. The disadvantage of this notation is its ineconomy: other functions (and the non-Ada like "function type") must be defined to achieve all meanings of the functional notation form for E. For example, E [open (E1,I2,I3,E4)] u is a function, not a value.

7.2.1 An example of Denotational Semantics for the Specification of Kernel Interfaces

We provide an example of the denotational approach to describe the kernel interfaces of CAIS package Node_Management. This example shows the beginning specification that must be specified for a denotational semantics: the domains Node and Asv, as well as most semantic clauses are left incomplete.

Kernel Facility: package Node_Management

Syntactic Domains

Ide	The domain of identifiers with elements I1,I2, ...
Exp	The domain of expressions with elements E1, E2,...
Com	The domain of commands with elements C1, C2,...
Dec	The domain of declarations with elements D1,D2,...

Syntactic Clauses

```
C ::=  open (E1,I2,I3,E4);
       | ciose (I1,I2,I3,I4);
       | change_intent (I1,I2,E3);
       | copy_node(I1,I2,I3,I4);
       | copy_tree (I1,I2,I3,I4);
       | rename (I1,I2,I3,E4);
       | link (E1,E2);
       | iterate (I1,I2,I3,E4,E5,E6);
       | get_next (I1,I2);
       | set_current_node (E1,E2);
       | get_current_uode (I1);

E ::=  is_open (I1)
       | kind (I1)
       | primary_name (I1)
       | primary_key (I1)
       | primary_relation (I1)
       | path_key (I1)
       | path_relation (I1)
       | obtainable (I1, I2, E3)
       | more (E1)
       | is_same (E1,E2)

D ::=    I1: node_iterator ;
       | I1: relationship_key_pattern := E1;
       | I1: relation_name_pattern := E1;
```

Semantic Domains

Env The domain of environments with elements u:

Env = Ide -> [Dv + {unbound}]

Dv The domain of denotable values with elements d:

Dv = Loc + Asv + Cc (Exceptions are denotable.)

Loc The domain of locations with elements l.

Asv The domain of assignable values with elements a.

Store The domain of stores with elements s:
Store = Loc -> [Sv + {unused}]

Sv The domain of storable values with elements v:
Sv = Node + Asv

Node The domain of nodes with elements n.

Cc The domain of command continuations with elements c:
Cc = Store -> Store

Ec The domain of expression continuations with elements k:
Ec = Dv -> Cc

Dc The domain of declaration continuations with elements d:
Dc = Env -> Cc

Semantic functions

Semantics of expressions:

E: Exp -> Env -> Ec -> Cc

Semantics of commands:

C: Com -> Env ->Cc ->Cc

Semantics of declarations:

D: Dec -> Env -> Dc -> Cc

Semantic Clauses (some examples)

Commands

C [open (E1,I2,I3,E4)] u c = {meaning}

Expressions

E [is_open (I1)] u k = {meaning}

Declarations

D [I1: node_iterator] u d = {meaning}

7.3. Analysis of Denotational Approach

The denotational approach to formal semantics can adequately specify kernel interfaces, provided one interprets these interfaces as defining a language. The complete specification of CAIS semantics for storage management and input/output can also be expressed, although it would be a laborious undertaking, even if aided by automated tools. The major tasks in these areas involves selecting a formal mathematical model for the CAIS data structures and devices. These formal models would then be represented in the notation chosen for the domains.

Semantics for process management can also be described in the denotational style, assuming that a formal model of concurrency (like Actor Semantics) is also similarly selected.

The denotational approach is not an alternative method to specifying semantics, rather, it emphasizes a different perspective toward specification. The denotational approach corresponds to a "top-down" solution to the problem of defining a language: the emphasis is on developing mathematical domains and functions to model machine meanings resulting from program execution. The operational approach corresponds to a "bottom-up" solution, whereby the emphasis is on constructing machine operations that will execute programs. An algebraic semantics is also a denotational semantics; in this approach, other specific mathematical constructs are used (more specific then domains) for representing the denotations. As observed above, a denotational specification becomes an operational specification if tools are provided that can "execute" the denotational semantic notation. Both approaches are used to construct rules of program properties to enable an axiomatic semantics.

It is not clear which approach is best for the specification of kernel interfaces. An operational approach would probably easier to understand (but harder to modify or check for consistency or completeness) then a denotational specification; conversely, a denotational specification is more amenable to a machine independent meaning. This last characteristic is important for achieving interoperability and transportability. On the other hand, the use of denotational semantics for the specification of concurrent computation in Ada has not been as adequately addressed as in some other languages; this implies that for process management, at least, many researchers are more comfortable with an operational approach.

8. CONCLUSIONS AND RECOMMENDATIONS

We have described how several semantic description techniques would be applied to a set of kernel facilities, using CAIS as an example. Considering informal methods, such as English narrative and example use, we have shown that these techniques are most useful during the developmental stage. They are quickly prepared and easily comprehended, which are important criteria for design reviews. The techniques lack in that it is easy to prepare descriptions that don't adequately treat details and are ambiguous. One recommendation is to explore a narrative description that is developed in close conjunction with a formal description. By doing so, the resulting description would be precise and nearly complete, as provided by the use of a formal definition as a basis. Further, the result would be more comprehensible than the formal specification.

Abstract Machine, Axiomatic and Denotational descriptions of kernel facilities have also been studied. These techniques have all been found to contain strengths and weaknesses with respect to the task at hand. The Abstract Machine description we presented, while comprehensible to the Ada community, lacks in applicabiliy to other sets of interfaces. Further, the reader of Abstract Machine programs is tempted to infer a single implementation technique. It is all too easy to adopt the techniques used in the Abstract Machine. The primary advantages of the Abstract Machine descriptions presented are:

1. All sections of the CAIS are equally well described. This is an attribute that is not shared with other methods.
2. The technique lends to an early and complete operational definition.
3. Although the description is not formal, it defines the CAIS in terms of the Ada langauge; thus centralizing related products.

An axiomatic description of the node management faciltiy COPY_NODE is presented in the paper as an example. It demonstrates an application ameniable to axiomatic description. With few exceptions, an axiomatic description of the Node Management section of CAIS provides a straightforward semantics. As noted, it is difficult to describe exception status returns and constructions allowing aliases Axiomatically. To describe the process control facilities of CAIS, additional formalism is needed. Additionally, an axiomatic description of input/output facilities would be bulky. The Axiomatic method does, however, lend itself to proving properties of programs using CAIS facilities.

Adaptation of denotational semantics to CAIS is also straightforward for the Node Management facilities. Existing denotational mechanisms can be applied directly from denotational descriptions of programming langauges. Again with this approach, input/output and process control present the greatest challenge to a concise denotational description.

9. REFERENCES

[FRE-82] Freedman, R.S. "A formal approach to APSE portability", *in The Public Report of the KIT/KITIA*, Vol 1., NOSC Technical Report, 1982.

[HOA-69] Hoare, C.A.R. "An axiomatic basis for computer programming", *Communications of the ACM* , Vol. 12, No. 10, (Oct. 1969) pp.576-83.

[HOA-73] Hoare, C.A.R. and Wirth, N. "An axiomatic definition of the programming language Pascal", *Acta Informatica*, Vol. 2, pp. 335-55.

[KAF-82] Kafura, D., Lee, J.A.N.; Lindquist,T.E. and Probert, T. "Validation in Ada programming support environments", Technical Report Department of Computer Science, CSIE-82-12, Virginia Tech Blacksburg Virginia.

[LIN-84] Lindquist,T.E. and Facemire,J.L., "A specification technique for the common APSE interface set", *Journal of Pascal, Ada and Modula-2* , Sept/Oct.

[LIN-85] Lindquist,T.E. and Facemire,J.L. "Using an Ada-based abstract Machine description of CAIS to generate validation tests", *proceedings of the Washington Ada Symposium* , ACM, March 1985.

[LON-78] London,R.L.; et.al "Proof rules for the programming language Euclid", *Acta Informatica* Vol. 10, pp. 1-26.

[SCO] Scott, D. and C. Strachey, "Towards a mathematical semantics for computer programs", *Proc. Symp. on Computers and Automata, Polytechnic Institute of Brooklyn*; also Tech. Mon. PrRG-6, Oxford U. Computing Lab., pp. 19-46.

[SRI-85] Srivastava,C.S. and Lindquist,T.E., "An abstract machine specification of the processnode section of CAIS", *proceedings of the Annual National Conference on Ada Technology* , Houston Texas, March 1985.

[YEL-84] Yelowitz,L. "Toward a formal semantics for the CAIS", *Public Report of the KIT/KITIA* , Vol. III, 1984.

Bjorner,D., Jones, C., Formal Specification and Software Development, Prentice-Hall International Series in Computer Science, 1982.

Clinger, W., Foundations of Actor Semantics, AI-TR-633, MIT Artificial Intelligence Laboratory, May, 1981.

Clemmensen, G., Oest, O., "Formal Specification of an Ada Compiler- A VDM Case Study," Dansk Datamatik Center, 1983-12-31, 1985.

Freedman, R.S., Programming with APSE Software Tools, "Chapter 5: Addendum: Formal Semantics," Petrocelli Books, Inc., Princeton, 1985.

Freedman, R.S., "Specifying KAPSE Interface Semantics," in Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report Volume II (P. Oberndorf, ed.), NOSC TD 552, October, 1982.

Gordon, M., The Denotational Description of Programming Languages: An Introduction, Springer-Verlag, New York 1979.

INRIA, Formal Definition of the Ada Programming Language (Preliminary Version for Public Review), Ada Joint Program Office, November 1980.

Kini, V., Martin, D., Stoughton, A., Tools for Testing the Denotational Semantic Definitions of Programming Languages, ISI/RR-83-112, USC, May, 1983.

Mosses, P., "Compiler Generation Using Denotational Semantics, in Lecture Notes in Computer Science, Vol. 45, Springer-Verlag, New York, 1976.

QUALITY ASSURANCE GUIDELINES

prepared for

KIT/KITIA

COMPLIANCE WORKING GROUP
(COMPWG)

SECOND DRAFT

April 14,1986

by
Lloyd Stiles
Code 844
FCDSSA San Diego

1. Introduction.

Software Quality Assurance (SQA) is a means for program managers to ensure the development and life-cycle maintenance of high quality computer programs. In the development of large scale computer systems, the guarentee of a high quality software systems requires a continuance of comprehensive reviews. These reviews ensure well defined requirements, specifications, design and code. Upon successful completion of each review cycle, the appropriate products are baselined and identified as configuration items (CI). The CIs are placed under configuration control and require formal procedures to implement any changes. The purpose of strong configuration control is to further ensure that the quality built in through top-down design is not degraded by uncontrolled changes. SQA can be visualized as a management umbrella under which the activities of quality control (QC), configuration management (CM) and evaluation and validation (E&V) are carried out. An SQA officer can act as the program manager's coordinator who maintains a system perspective of the entire software development and establishes a system of independent checks and balances in the form of reviews and audits to verify the quality at each phase of development. While it is recognized that excessive SQA activities can kill any project by stifling productivity; too little SQA can allow a poor or unusable program to be produced.

Figure 1 illustates the documentation scheme currently used at FCDSSA San Diego. This is based on using MIL-STD-490 for the type "A" system specification and DOD-STD-1679 for all other products. The documents have been divided into three functional areas - software management, software validation (testing) and software development. The series of reviews and audits as well as the configuration status accounting are the activities and reports required to support software quality assurance.

Although not always realized by the software engineers, designers and programmers, implementation of management disciplines into a software project are as important as the engineering disciplines. Figure 2a and 2b illustate the first three level breakdown for a software project. Figures 3 and 4 expand the remaining components required for QC and CM respectively.

The resources to support software engineering and management varies by project/program and depends on the following factors:

a. The size and complexity of the development effort.

b. The development methodology implemented.

c. The anticipated computer program's life-cycle.

d. The extent of the computer program's visibility and usage.

e. The availability of applicable automated support tools or systems.

f. The requirements and constaints directed by higher authority.

g. The budgetary constaints imposed.

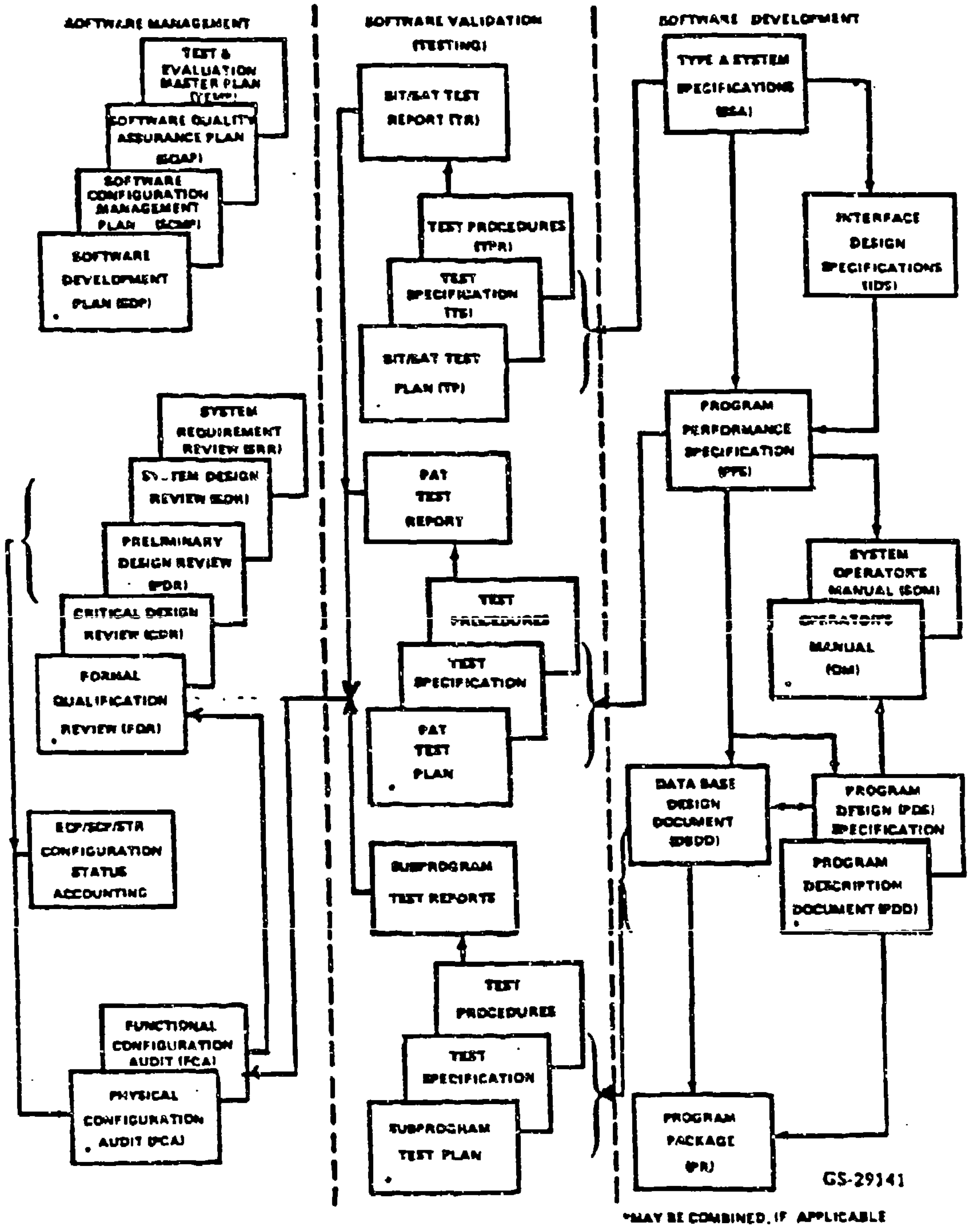

SOFTWARE MANAGEMENT
TEST & EVALUATION MASTER PLAN (TEMP)
SOFTWARE QUALITY ASSURANCE PLAN (SQAP)
SOFTWARE CONFIGURATION MANAGEMENT PLAN (SCMP)
SOFTWARE DEVELOPMENT PLAN (SDP)
SYSTEM REQUIREMENT REVIEW (SRR)
SYSTEM DESIGN REVIEW (SDR)
PRELIMINARY DESIGN REVIEW (PDR)
CRITICAL DESIGN REVIEW (CDR)
FORMAL QUALIFICATION REVIEW (FQR)
ECP/SCP/STR CONFIGURATION STATUS ACCOUNTING
FUNCTIONAL CONFIGURATION AUDIT (FCA)
PHYSICAL CONFIGURATION AUDIT (PCA)
SOFTWARE VALIDATION (TESTING)
BIT/SAT TEST REPORT (TR)
TEST PROCEDURES (TPR)
TEST SPECIFICATION (TS)
BIT/SAT TEST PLAN (TP)
FAT TEST REPORT
TEST PROCEDURES
TEST SPECIFICATION
FAT TEST PLAN
SUBPROGRAM TEST REPORTS
TEST PROCEDURES
TEST SPECIFICATION
SUBPROGRAM TEST PLAN
SOFTWARE DEVELOPMENT
TYPE A SYSTEM SPECIFICATIONS (SSA)
INTERFACE DESIGN SPECIFICATIONS (IDS)
PROGRAM PERFORMANCE SPECIFICATION (PPS)
SYSTEM OPERATOR'S MANUAL (SOM)
OPERATOR'S MANUAL (OM)
DATA BASE DESIGN DOCUMENT (DBDD)
PROGRAM DESIGN (PDS) SPECIFICATION
PROGRAM DESCRIPTION DOCUMENT (PDD)
PROGRAM PACKAGE (PP)
GS-29341
*MAY BE COMBINED, IF APPLICABLE

Figure 1. Sample Documentation Scheme

```
Software      << Engineering<< Requirement Analyzers
Engineering   ¦    Support     ¦ Rapid Prototyping
              ¦                ¦ Program Design Language(ie:Ada BYRON)
              ¦                ¦_Ada Program Support Environment
              ¦    Software   << Requirement          << System Reqmts
              ¦    Life Cycle  ¦                        ¦ Software Reqmts
              ¦    Phases      ¦                        ¦_Management Plans
              ¦                ¦ Specification        << Software Specs
              ¦                ¦                        ¦_Test Plan
              ¦                ¦ Design               << Program Specs
              ¦                ¦                        ¦ Data Base Specs
              ¦                ¦                        ¦_Test Specs
              ¦                ¦ Development          << Program Coding
              ¦                ¦                        ¦ Program Integration
              ¦                ¦                        ¦_Test Procedure Coding
              ¦                ¦ Validation           << Unit Testing
              ¦                ¦                        ¦ Subprogram Testing
              ¦                ¦                        ¦ Integration Testing
              ¦                ¦                        ¦ Performance Testing
              ¦                ¦                        ¦_System Testing
              ¦                ¦_Maintenance          << STR Correction
              ¦                                         ¦_ECP/SCP Design
              ¦ Software      << Memorandums
              ¦ Engineering    ¦ Correspondance
              ¦_Notes          ¦ Meeting Minutes
                               ¦ White Papers
                               ¦_Studies and Reports
 Training     << Curriculum
               ¦ Lesson Plans
               ¦ Instruction Guides
               ¦_Implementation Media << Classroom Instruction
                                       ¦ Mockup/Simulation
                                       ¦ Programmed Instruction
                                       ¦_Video Presentation
```

Figure 2a. Software Project Component Structure (Engineering/Training)

```
Software    << Software       << Resources      << Time
Management   ¦ Development     ¦                 ¦ Funding
             ¦ Planning        ¦                 ¦ Personnel
             ¦                 ¦                 ¦ Facilities
             ¦                 ¦ Schedules      << PERT
             ¦                                   ¦ CPA
             ¦                                   ¦ Gantt
Quality     << Quality        << Administration << Quality ~~Assurance~~ Control
Assurance    ¦ Control         ¦                 ¦     Office
             ¦                 ¦ Evaluation
             ¦                 ¦ ~~Verification~~   << Document Review
             ¦                 ¦                 ¦ Code Review
             ¦                 ¦                 ¦ ECP/SCP Review
             ¦                 ¦ Monitoring     << Configuration Management
             ¦                                   ¦     Activities
             ¦                                   ¦ Testing Activities
             ¦ Configuration  << Administration << Configuration Management
             ¦ Management      ¦                 ¦     Office
             ¦                 ¦                 ¦ Configuration Control
             ¦                 ¦                 ¦     Boards
             ¦                 ¦ Identification << Baselines
             ¦                 ¦                 ¦ Libraries
             ¦                 ¦                 ¦ Status Accounting
             ¦                 ¦ Control        << Baselines
             ¦                 ¦                 ¦ Libraries
             ¦                 ¦ Status         << Report Generation
             ¦                 ¦   Accounting
             ¦                 ¦
             ¦                 ¦ Audits         << Physical Configuration
             ¦                                   ¦ Functional Configuration
             ¦ Evaluation and << ** The testing components are listed
             ¦ Validation (E&V)¦ in the Validation phase of the
                               ¦ Software Life Cycle
```

Figure 2b. Software Project Component Structure (Management)

2. General Concepts.

Guidelines to implement quality ~~control & test~~ ~~assurance~~ assurance into any software development involves:

a. Integrating the software engineering and software management components into a smooth project flow.

b. Prioritizing, selecting and implementing the ~~optimal quality~~ applicable evaluation factors and criter into the project.

c. Identifying and approving all applicable standards, guidelines and criteria before their required implementation.

d. Providing definitive statements of work (SOW) for all development effort.

e. Providing configuration identification for all products slated for configuration control.

f. Providing definitive SOW's for all formal reviews and configuration audits and assigning the necessary responsibility and authority.

g. Establishing an appropriate baseline concept and allocating configuration items (CI)s to each baseline.

h. Establishing or identifying a suitable configuration status accounting system.

i. Establishing or identifying document and program library facilities and assigning each CI to the appropriate library.

j. Establishing configuration control boards, identifying the members and defining each board and member's duties.

k. Providing for periodic monitoring of configuration management and testing activities.

l. Providing time, personnel and funding resources.

Quality Control	Administration	Quality ~~Assurance~~ Control Office	
	Evaluation ~~Verification~~ (informal, formal, in-progress)	Document Review	Applicable Evaluation Criteria *
		Code Review	Programming Conventions Error Free Compile Applicable Evaluation Criteria *
		STR/ECP/SCP Review/Audit	System Perspective Applicable Evaluation Criteria *
	~~Evaluation/~~ Certification	Predevelopment Baselines Support Software/ Tools	Physical Configuration Audit (PCA) Functional Configuration Audit (FCA)
	Program Monitoring	Configuration Management Activities Testing Activities	

* Evaluation:

Factors	Criteria	Factors	Criteria
Operational	Feasibility	Maintainability	Consistency Simplicity Conciseness Modularity Self-descriptiveness
Technical	Preciseness	Testability	Simplicity Modularity Instrumentation Self-descriptiveness
Standards	Compliance. Tech Editing	Portability	Modularity Self-descriptiveness Hardware Independence Software Indepenndence
Correctness	Traceability Consistency Completeness	Reusability	Generality Modularity Software Independence Hardware Independence Self-descriptiveness
Reliability	Error Tolerance Consistency Accuracy Simplicity	Interoperability	Modularity Comunication Commonality Data Commonality
Security	Data Integrity Process Integrity	Efficiency	Execution Efficiency Storage Efficiency
Usability	Training Communicativeness Operability	Reproduceability	Code Validity
Flexibility	Modularity Generality Self-descriptiveness Expandability		

Figure 4. Quality ~~Assurance~~ Control Component Structure

```
Software        << Administration    << Configuration Management Office
Configuration    ¦                    ¦_Configuration Control Boards
Management       ¦ Configuration     << Baselines     << Requirements
                 ¦ Identification     ¦                ¦ Specification
                 ¦                    ¦                ¦ Allocated
                 ¦                    ¦                ¦ Product
                 ¦                    ¦                ¦_Operational
                 ¦                    ¦ Libraries      ¦ Application Program
                 ¦                    ¦                ¦ Support Program
                 ¦                    ¦                ¦ Document
                 ¦                    ¦                ¦_Project Data Base
                 ¦                    ¦ Status        << Trouble Reports
                 ¦                    ¦_ Accounting    ¦ Engineer Change Proposal
                 ¦                                     ¦_Software Change Proposal
                 ¦ Configuration     << Baseline      << Formal Reviews
                 ¦ Control            ¦ Control        ¦_Configuration Audits
                 ¦                    ¦
                 ¦                    ¦ Library       << Application Program
                 ¦                    ¦ Control        ¦ Support Program
                 ¦                    ¦                ¦ Documentation
                 ¦                    ¦                ¦_Project Data Base
                 ¦                    ¦ Change        << Trouble Reports
                 ¦                    ¦_Control        ¦ Engineer Change Proposal
                 ¦                                     ¦_Software Change Proposal
                 ¦ Configuration     << Report        << Program Configuration
                 ¦_Status Accounting  ¦__ Generator    ¦     Status Report
                                                       ¦ ECP/SCP Status Report
                                                       ¦_STR Status Report
================================================================================

Application     << Application Process Source/Object Code
Program          ¦ Application Database Source/Object Code
Library          ¦ RTAS Source/Object Code (Target)
                 ¦ RTOS Source/Object Code (Target)
                 ¦_Test Program Source/Object Code
Support         << Operating System (Host)
Program          ¦ Ada Language System/Army
Library          ¦ Integrated Software Environment(Ada)/AF
                 ¦ Ada Language System/Navy
                 ¦ Status Accounting System
                 ¦ Byron(Ada)
                 ¦ Problem Statement Analyzer (PSA)
                 ¦_Requirements Engineering Validation System (REVS)
Documentation   << Application Development Documents
Library          ¦ Support Development Documents
                 ¦ Project Management Documents
                 ¦ Test Documents
                 ¦_Review/Audit Reports
Project         << Application Requirement/Design Perameters (PSA/REVS)
Data Base        ¦_Status Accounting
```

Figure 5. Configuration Management Component Structure

3. ~~Quality Factor~~ Evaluation Factor Concepts.

The SQA components illustrated in figure 3 provide a list of ~~quality~~ evaluation factors and the corresponding ~~quality~~ evaluation factor criteria. The concept of ~~quality~~ evaluation factors is based on an extensive Air Force study; the reports have been republished by DOD. Table 1 provides definitions for quality factors; table 2 provides definitions for the related quality factor criteria. While there are many other terms to describe software quality, this document provides a definitive list. Implementing this concept allows a more disciplined engineering approach to software quality assurance. The software program manager is provided with conceptually simple, easy to use procedures for specifying required quality in more precise terminology. The PM essentially performs a trade-off analysis for the requirements of the system. The software developer is forced to address how they plan to build the required quality into the software. Specific software quality attributes required are independent of the design and implementation techniques used. Implementation of the applicable quality factors' criteria provides the PM with a quantifiable criteria against which to judge the software quality prior to acceptance testing and operational use.

In establishing comprehensive reviews it is necessary to go beyond quality factors. Reviews must also include operational and technical evaluation and the conformance to applicable standards. While it usually impossible to find a single person capable of the full comprehensive review of a program, it is possible to have several people review from his or her area of expertise. Thus a subjective review by a single person becomes more objective when reviewed by several people. Further refinement is possible by implementing checkoff lists for each product being reviewed and for each applicable criteria.

4. Summary.

Providing quality software products is accomplished by:

a. Developing product quality through definitive statements of work.

b. Verifying product quality through comprehensive reviews.

c. Validating product quality through thorough testing.

d. Maintaining product quality through stringent configuration control.

FACTORS	DEFINITIONS
CORRECTNESS	Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
RELIABILITY	Extent to which a program can be expected to perform its intended function with required precisions.
EFFICIENCY	The amount of computing resources and code required by a program to perform a function.
INTEGRITY	Extent to which access to software or data by unauthorized persons can be controlled.
USABILITY	Effort required to learn, operate, prepare input, and interpret output of a program.
MAINTAINABILITY	Effort required to locate and fix an error in an operational program.
FLEXIBILITY	Effort required to modify an operational program.
TESTABILITY	Effort required to test a program to ensure it performs its intended function.
PORTABILITY	Effort required to transfer a program from one hardware configuration and/or software system environment to another.
REUSABILITY	Extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform.
INTEROPERABILITY	Effort required to couple one system with another.

Table 1. Software Quality Factor Definitions.

CRITERION	DEFINITIONS
TRACEABILITY	Those attributes of the software that provide a thread from the requirements ~~to~~ the implementation with respect to the specific development and operational environment.
COMPLETENESS	Those attributes of the software that provide full implementation of the functions required.
CONSISTENCY	Those attributes of the software that provide uniform design and implementation techniques and notation.
ACCURACY	Those attributes of the software that provide the required precision in calculations and outputs.
ERROR TOLERANCE	Those attributes of the software that provide continuity of operation under adverse operating conditions.
SIMPLICITY	Those attributes of the software that provide implementation of functions in the most understandable manner. (Usually avoidance of practices which increase complexity.)
MODULARITY	Those attributes of the software that provide a structure of highly independent modules.
GENERALITY	Those attributes of the software that provide breadth to the functions performed.
EXPANDABILITY	Those attributes of the software that provide for expansion of data storage requirements or computational functions.
INSTRUMENTATION	Those attributes of the software that provide for the measurement of usage or identification of errors.
SELF DESCRIPTIVENESS	Those attributes of the software that provide explanation of the implementation of a function

Table 2. Criteria Definitions for Software Quality

CRITERION	DEFINITIONS
EXECUTION EFFICIENCY	Those attrributes of the software that provide for minimum processing time.
STORAGE EFFICIENCY	Those attributes of the software that provide for minimum storage requirements during operation.
ACCESS CONTROL	Those attributes of the software that provide for control of the access of software and data.
ACCESS AUDIT	Those attributes of the software that provide for an audit of the access of software and data.
OPERABILITY	Those attributes of the software that determine operation and procedures concerned with the operation of the software.
TRAINING	Those attributes of the software that provide transition from current operation or initial familiarization.
COMMUNICATIVENESS	Those attributes of the software that provide useful inputs and outputs which can be assimilated.
SOFTWARE SYSTEM INDEPENDENCE	Those attributes of the software that determine its dependency on the software environment (operating systems, utilities, input/output routines, etc.)
MACHINE INDEPENDENCE	Those attributes of the software that determine its dependency on the hardware system.
COMMUNICATIONS COMMONALITY	Those attributes of the software that provide the use of standard protocols and interface routines.
DATA COMMONALITY	Those attributes of the software that provide the use of standard data representations.
CONCISENESS	Those attributes of the software that provide for implementation of a function with a minimum amount of code.

Table 2. Criteria Definitions for Software Quality (cont)

Ada Interoperability Survey

The Kernel Ada Programmming Support Environment (KAPSE) Interface Team (KIT) / KAPSE Interface Team from Industry and Academia (KITIA) would like to collect data about the Ada interoperability problems that users of the Ada language and its support environments are experiencing. The attached form is provided for consistent and thorough collection of relevant data. The data collected will be used to prepare an Ada Interoperability Guide which will include descriptions of actual user problems as well as proposed solutions, work-arounds, and guidelines for promoting the interoperability of tools and APSEs.

The following examples demonstrate the types of problems KIT/KITIA is interested in documenting:

a. problems associated with moving data across APSEs,
b. problems associated with rehosting efforts,
c. incompatibilities between Ada compilers (e.g., creating a data file with a program generated from one compiler and not being able to read the data with a program generated from a different compiler on the same machine),
d. incompatibilities among other Ada tools, and
e. problems that may occur in multi-lingual environments (e.g. translating array indexing from arrays passed between Ada and and Fortran procedures).

Any reports of Ada interoperability problems will be appreciated. KIT/KITIA is also interested in related interoperability problems that do not directly involve Ada.

Please send the completed form to the following address:

F. Matthew Emerson
Naval Avionics Center, D/825
6000 E. 21st Street
Indianapolis, IN 46219.

A copy of the final version of the Ada Interoperability Guide will be mailed to the mailing address provided by each respondent who sends us at least one completed Ada Interoperability Problem Report Form.

Ada Interoperability Problem Report Form page 1 of 2

1. General --

1.1. Organization:

->

1.2. Class (General Nature) of Problem (one word or phrase, if possible):

->

1.3. Date of Problem:

->

2. Background ---

2.1. Computer, Operating System, and APSE (if applicable):
(include version numbers.)

->

2.2. Languages, Tools, and Programs Involved:
(include compilers and version numbers.)

->

2.3. Project Affiliation:

=>

3. Communication --

3.1. Name of Individual(s) Involved:

->

3.2. Mailing Address:

->

3.3. Telephone Number:

->

3.4. Net Address:

->

Interoperability Problem Report Form page 2 of 2

4. Detail --

4.1. Description of Problem:

->

4.2. Proposed or Working (circle) Solution:

->

Compiled - 5/28/85
Updated - 8/12/85

Updated - 11/29/85

DRAFT KIT/KITIA GLOSSARY

This glossary provides definitions for the terms used by the KIT/KITIA. Most terms are cited in the three KIT/KITIA documents listed below. Many of these definitions are very document-specific.

1. Proposed MIL-STD-CAIS, 31 January 1985.
2. DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS), 13 September 1985. (Better known as the RAC document.)
3. Guidelines and Conventions Working Group (GACWG) Transportability Guide, forthcoming.

The key below indicates which working group contributed the term.

KEY

[C] - Term is defined by the CAISWG and is cited in the Proposed MIL-STD-CAIS (Ref. 1 above).

[CP] - Term is defined by the COMPWG.

[R] - Term is defined by the RACWG and is cited in the RAC document (Ref. 2 above).

[G] - Term is defined by the GACWG and is cited in GACWG Transportability Guide (Ref 3 above).

No letter(s) and no brackets indicates that the word is a general KIT/KITIA term.

When a term is taken from another source, the source document is abbreviated within parentheses preceding the definition. Following the glossary is a list of the abbreviations and complete titles for these references.

TERM

DEFINITION
----

abort [C] -
(IEEE) To terminate a process prior to completion.

abstract machine [CP] -
TBD

access [C] -
(TCSEC) A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

access checking [C] -
The operation of checking access rights against those rights required for the intended operation, according to the access control rules, and either permitting or denying the intended operation.

access control [C] -
(TCSEC) (1) discretionary access control: A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject. (2) mandatory access control: A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. In the CAIS, this includes specification of access rights, access control rules and checking of access rights in accordance with these rules.

access control constraints [C] -
The resulting restrictions placed on certain kinds of operations access control.

access control information [C] -
All the information required to perform access checking.

access control rules [C] -
The rules describing the correlations between access rights and those rights required for an intended operation.

access relationship [C] -
A relationship of the predefined relation ACCESS.

access rights [C] -
Descriptions of the kinds of operations which can be performed.

access to a node [C] -
Reading or writing of the contents of the node, reading or writing of attributes of the node, reading or writing of relationships emanating from a node or of their attributes, and traversing a node as implied by a pathname.

accessibility [C] -
The subject has (adopted a role which has) been granteo the access right EXISTENCE to the object.

activate [R] -
To create a CAIS process. The activation of a program binds that program to its execution environment, which are the resources required to support the process's execution, and includes the program to be executed. The activation of a process marks the earliest point in time which that process can be referenced as an entity within the CAIS environment.

active position [C] -
The position at which a terminal operation is to be performed.

Ada Programming Support Environment (APSE) [C] -
(UK Ada Study, STONEMAN) A set of hardware and software facilities whose purpose is to support the development and maintenance of Ada applications software throughout its life-cycle with particular emphasis on software for embedded computer applications. The principal features are the database, the interfaces and the tool set.

adopt a role [C] -
The action of a process to acquire the access rights which have been or will be granted by an object to adopters of that role; in the CAIS this is accomplished by establishing a secondary relationship of the predefined relation ADOPTED_ROLE from the process node to the node representing the role.

adopted role of a process [C] -
The access rights associated with the node that is the target of a relationship of the predefined relation ADOPTED_ROLE emanating from the process node or with any group node one of whose permanent members is the target of such a relationship.

advance (of an active position) [C] -
(1) Scroll or page terminal: Occurs whenever (i) the row number of a new position is greater than the old or (ii) the row number of the new position is the same and the column number of the new position is greater than that of the old. (2) Form terminal: Occurs whenever the indices of its position are incremented.

approved access rights [C] -
Access rights whose names appear in resulting rights lists of relevant grant items for which either (i) the necessary right is null or (ii) the necessary right is an approved access right.

archive [R] -
A subset of the CAIS-managed data that has been relegated to backing storage media while retaining the integrity, consistency and availability of all information in the entity management system.

area qualifier [C] -
A designator for the beginning of a qualified area.

attribute [C] -
A named value associated with a node or a relationship which provides

information about that node or relationship.

attribute [R] -
An association of an entity or relationship with an elementary value.

axiomatic semantics [CP] -
Axiomatic semantics of the CAIS involves stating consistency conditions of states of the CAIS using predicate calculus in which these consistency conditions must hold for all states. Then, the effects of subprogram interfaces of the CAIS are specified using the predicate calculus to describe state changes brought about by invoking the given subprogram. Finally, it is proven that the consistency conditions remain invariant after invocation of any subprogram.

backup [R] -
A redundant copy of some subset of the CAIS-managed data. The subset is capable of restoration to active use by a CAIS implementation, particularly in the event of a loss of completeness or integrity in the data in use by implementation.

block data [R] -
A sequence of one or more data units which is treated as an indivisible group by a transmission mechanism.

block terminal [R] -
A terminal that transmits/receives a block of data units at a time.

block unit [R] -
A representation of a value of an Ada discrete type.

closed node handle [C]
A node handle which is not associated with any node.

completion [R] -
The voluntary termination of a process. Completion of a process is always self-determined.

consumer [R] -
An entity that is receiving data units via a datapath.

contents [C] -
A file or process associated with a CAIS node.

couple [C] -
To establish a correlation between a queue file and a secondary storage file. If the queue file is a copy queue file, its initial contents is a copy of the secondary storage file to which it is coupled; if the queue is a mimic queue file, its contents is a copy of the secondary storage file to which it is coupled, and elements that are written to the mimic queue file are appended to its coupled file.

current job [C] -
The root process node of the tree containing the current process node; represented by the predefined relation CURRENT_JOB.

current node [C] -
The node that is currently the focus or context for the activities of

the current process; represented by the predefined relation CURRENT_NODE.

current process [C] -
The currently executing process making the call to a CAIS operation. Pathnames are interpreted in the context of the current process.

current user [C] - The user's top-level node; represented by the secondary relationship of the predefined relation CURRENT_USER.

data unit [R] -
A representation of a value of an Ada discrete type.

datapath [R] -
The mechanism by which data units are transmitted from a producer to a consumer.

datastream [R] -
The data units flowing from a producer to a consumer (without regard to the implementing mechanism).

deactivate [R] -
To remove a terminated process so that it may no longer be referenced within the CAIS environment.

denotational semantics [CP] -
TBD

descendant (of a node) [C] -
Any node which is reachable from other nodes via primary relationships.

dependent process [C] -
A process other than a root process.

device [C] -
(WEBS) A piece of equipment or a mechanism designed to serve a special purpose or perform a special function.

device name [C] -
The keys of a primary relationship of the predefined relation DEVICE.

discretionary access control [C] -
See access control.

element (of a file) [C] -
A value of the generic data type with which the input and output package was instantiated; see [LRM] for additional information.

elementary value [R] -
One of two kinds of representations of data: interpretec and uninterpreted.

encapsulation [G] -
Encapsulation means placing procedures, functions, exceptions, types, etc. that all pertain to the same object into one package.

end position [C] -
The position of a form identified by the highest row and column indices

of the form.

entity [R] -
A representation of a person, place, event or thing.

exact identity [R] -
A designation of an entity (or relationship) that is always associated with the entity (or relationship) that it designates. This exact identity will always designate exactly the same entity (or relationship), and it cannot be changed.

extensible [R] -
Designed to facilitate development and use of portable extensions; reuseable to facilitate combination to create new interfaces and facilities which are also portable.

external file [C] -
(LRM 14.1.1 - Ada external file) Values input from the external environment of the program, or output to the environment, are considered to occupy external files. An external file can be anything external to the program that can produce a value to be read or receive a value to be written.

file [C] -
See external file.

file handle [C] -
An object of type FILE_TYPE which is used to identify an internal file.

file node [C] -
A node whose contents are an Ada external file, e.g., a host system file, a device, or a queue.

form [C] -
A form is a two-dimensional matrix of character positions.

group [C] -
A collection of nodes representing roles and identified by a structural node with emanating relationships of the predefined relations POTENTIAL_MEMBER and PERMANENT_MEMBER identifying each of the group's members. A member may be a user top-level node; a node representing the executable image of a program, or a node representing a group.

hardcopy terminal [R] -
A terminal which transmits/receives one data unit at a time and does not have an addressable cursor.

history [R] -
A recording of the manner in which entities, relationships and attribute values were produced and of all information which was relevant in the production of those entities, relationships or attribute values.

host -
(K/K) A host is a computer upon which an APSE resides, executes and supports Ada software development and maintenance. Examples are IBM 360/370, VAX 11/780, CDC 6000/7000, DEC 20 and UNIVAC 1110 with any of their respective operating systems.

identification [R] -
A means of specifying the entities, relationships and attributes to be operated on by a designated operation.

illegal identification [C] -
A node identification in which the pathname or the relationship key or relation name is syntactically illegal with repect to the syntax defined in Table 1 (of proposed MIL-STD-CAIS, 31 January 1985).

inaccessible [C] -
The subject has not (adopted a role which has) been granted the access right of EXISTENCE to the object.

initiate [C] -
To place a program into execution; in the CAIS, this means a process node is created, a process is created as its contents, required resources are allocated, and execution is started.

initiated process [C] -
The process whose program has been placed into execution.

initiating process [C] -
The process placing a program into execution.

isolation [G] -
Isolation means hiding a particular implementation in the package body. Typically isolated packages do not depend on other packages (they stand alone).

interface [C] -
(DACS) A shared boundary.

internal file [C] -
A file which is internal to a CAIS process. Such a file is identified by a file handle.

interoperability -
(K/K) Interoperability is the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Interoperability is measured in the degree to which this exchange can be accomplished without conversion.

interpreted data [R] -
A data representation whose structure is controlled by CAIS facilities and may be used in the CAIS operations. Examples are representations of integer, string, real, date and enumeration data, and aggregates of such data.

iterator [C] -
A variable which provides the bookkeeping information necessary for iteration over nodes (a node iterator) or attributes (an attribute iterator).

job [C] -
A process node tree, spanned by primary relationships, which develops under a root process node as other(dependent) processes are initiated for the user.

KAPSE -
(UK Ada Study) That level of an APSE which provides database communication and runtime support functions to enable the execution of an Ada program (including a MAPSE tool) and which presents a machine-independent portability interface.

key [C] -
See relationship key. The key of a node is the relationship key of the last element of the node's pathname.

label group (of a magnetic tape) [C] -
One of the following: (i) a volume header and af file header label, (ii) a file header label, or (iii) and end-of-file label.

latest key [C] -
The final part of a key that is automatically assigned lexicographically following all previous keys for the same relation names and initial relationship key character sequence for a given node.

list [C] -
(IEEE) An ordered set of items of data; in the CAIS, an entity of LIST_TYPE whose value is a linearly ordered set of data elements.

list item [C] -
A data element in a list.

mandatory access control [C] -
See access control.

MAPSE -
(UK Ada Study, STONEMAN) That level of an APSE which provides a minimal set of tools, written in Ada and supported by the KAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs. The term is used in this [UK] study to mean not a strictly minimal set, but a set with which a user can happily work.

modular [R] -
Designed such that it may be understood in isolation and such that there are no hidden interactions.

named item [C] -
A list item which has name associated with it.

named list [C] -
A list whose items are all named.

node [C] -
A representation within the CAIS of an entity relevant to the APSE.

node handle [C] -
An Ada object of type NODE_TYPE which is used to identify a CAIS node; it is internal to a process.

non-existing node [C] -
A node which has never been created.

object [C] -
(TCSEC) A passive entity that contains or receives information. In the CAIS, any node may be an object.

obtainable [C] -
A node is obtainable if it is created and not deleted.

open node handle [C] -
A node handle that has been assigned to a particular node.

operational semantics [CP] -
TBD

page terminal [R] -
A terminal which transmits/receives one data unit at a time and has an addressable cursor.

parent [C] -
The source node of a primary relationship; also the target of a relationship of the predefined relation PARENT.

path [C] -
A sequence of relationships connecting one node to another. Starting from a given node, a path is followed by traversing a sequence of relationships until the desired node is reached.

path element [C] -
A portion of a pathname representing the traversal of a single relationship; a single relation name and relationship key pair.

pathname [C] -
A name for a path consisting of the concatenation of the names of the traversed relationship in the path in the same order in which they are traversed.

permanent member [C] -
A group member which is intrinsically related to the group via primary relationships of the predefined relation PERMANENT_MEMBER.

position (of a terminal) [C] -
A place in an output device in which a single, printable ASCII character may be graphically displayed.

potential member [C] -
A group member that may dynamically acquire membership in the group; represented by a node that is the target of a secondary relationship of the predefined relation POTENTIAL_MEMBER emanating form that group node or form any of that group nodes' descendants.

pragmatics [C] -
Contraints imposed by an implementation that are not defined by the syntax or semantics of the CAIS.

primary relationship [C] -
The initial relationship established from an existing node to a newly created node during its creation. The existence of a node is determined by the existence of the primary relationship of which it is the target.

process [C] -
The execution of an Ada program including all its tasks.

process [R] -
The CAIS facility used to represent the execution of any program.

process node [C] -
A node whose contents represent a CAIS process.

producer [R] -
An entity that is transmitting data units via a datapath.

program [C] -
(LRM) A program is composed of a number of compilation units, one of which is a subprogram called the main program.

program [R] -
A set of compilation units, one of which is a subprogram called the "main program." Execution of the program consists of execution of the main program, which may invoke subprograms declared in the compilation units of the program.

qualified area [C] -
A contiguous group of positions in a form that share a common set of characteristics.

queue [C] -
(IEEE) A list that is accessed in a first-in, first-out manner.

rehostability -
(K/K) Rehostability of an APSE is the ability of the APSE to be installed on a different HOST. Rehostability is measured in the degree to which the installation can be accomplished with needed re-programming localized to the KAPSE. Assessment of rehostability includes any needed changes to non-KAPSE components of the APSE, in addition to the changes to the KAPSE.

relation [C] -
In the node model, a class of relationships sharing the same name,

relation name [C] -
The string that identifies a relation.

relationship [C] -
In the node model, an edge of the directed graph which emanates from a source node and terminates from a target node. A relationship is an instance of a relation.

relationship [R] -
An ordered connection or association among entities. A relationship among N entities (not necessarily distinct) is known as an "N-ary" relationship.

relationship key [C] -
The string that distinguishes a relationship from other relationships having the same relation name and emanating from the same node.

relevant grant items [C] -
The items in values of GRANT attributes of relationships of the relation ACCESS emanating from the object and pointing at any node representing a role which is an adopted role of the subject or representing a group, one of whose permanent members is an adopted role of the subject.

resource [R] -
Any capacity which must be scheduled, assigned, or controlled by the operating system to assure consistent and non-conflicting usage by programs under execution. Examples of resources include: CPU time, memory space (actuals and virtual), and shared facilities (variables, devices, spoolers, etc.).

resume [R] -
To resume the execution of a suspended process.

retargetablity -
(K/K) Retargetability is the ability of a target-sensitive APSE tool to accomplish the same function with respect to another target. Retargetability is measured in the degree to which this can be accomplished without modifying the tool. Not all tools will have target specific functions.

reusability -
(K/K) Reusability is the ability of a program unit to be employed in the design, documentation, and construction of new programs. Reusability is measured in the degree to which this reuse can be accomplished without reprogramming.

role [C] -
A set of access rights that a subject can acquire.

root process node [C] -
The initial process node created when a user logs on to an APSE or when a new job is created via the CREATE_JOB interface.

secondary relationship [C] -
An arbitrary connection which is established between two existing nodes.

security -
The management, protection, and distribution of information.

security level [C] -
(TCSEC) The combination of a hierarchical classification and a set of non-hierarchical categories that represents the sensitivity of information.

security policy -
The set of laws, rules, and relationships that regulate how an organization manages, protects, and distributes information.

semantics [CP] -
TBD

"shall" [R] -
Indicates a requirement on the definition of the CAIS; sometimes

"shall" is followed by "provide" or "support", in which cases the following two definitions supersede this one.

"shall provide" [R] -
Indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities.

"shall support" [R] -
Indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities or for CAIS definers to demonstrate that the capability may be constructed from CAIS interfaces.

"should" [R] -
Indicates a desired goal but one for which there is no objective test.

source node [C] -
The node from which a relationship emanates.

start position (of a form terminal) [C] -
The position of a form identified by row one, column one.

structural node [C] -
A node without contents. Structural nodes are used strictly as holders of relationships and attributes.

subject [C] -
(TCSEC) An active entity, generally in the form of a person, process, or device, that causes information to flow among objects or changes the system state. In the CAIS, a subject is always a process.

suspend [R] -
To stop the execution of a process such that it can be resumed. In the context of an Ada program being executed, this implies the suspension of all tasks, and the prevention of the activation of any task until the process is resumed. It specifically does not imply the release of any resources which a process has assigned to it, or which it has acquired, to support its execution.

system-level node [C] -
The root of the CAIS primary relationship tree which spans the entire node structure.

target -
(K/K) A target is a computer system upon which Ada programs execute. Remark: Hosts are, in fact, also targets, in as much as the APSE is written in Ada. A target might not be capable of supporting an APSE. An embedded target is a target which is used in mission critical applications. Examples of embedded target computer systems are AN/AYK-14, AN/UYK-43 and computer systems conforming to MIL-STD-1750A and MIL-STD-1862 (NEBULA).

target node [C] -
The node at which a relationship terminates.

task [C] -
(LRM) A task operates in parallel with other parts of the program.

task wait [R] -
Delay of the execution of a task within a process until a CAIS service requested by this task has been performed. Other tasks in the same process are not delayed.

terminal [R] -
An interactive input/output device.

terminate [R] -
To stop the execution of a process such that it cannot be resumed.

termination of a process [C] -
Termination (see [LRM] 9.4) of the execution of the subprogram which is the main program (see [LRM] 10.1) of the process.

token [C] - An internal representation of an identifier which can be manipulated as a list item.

tool [C] -
(IEEE - software tool) A computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, an automated design tool, compiler, test tool, or maintenance tool.

top-level node [C] -
A structural node representing the user. Each user has a top-level node.

traceability [CP] -
TBD

track [C] -
(1) An open node handle is guaranteed always to refer to the same node, regardless of any changes to relationships that could cause pathnames to become invalid or to refer to different nodes. An open node handle is said to track the node to which it refers. (2) Secondary relationships.

transportability [G] -
(K/K) Transportability of an APSE tool is ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonyms.

traversal of a node [C] - Traversal of a relationship emanating from the node.

traversal of a relationship [C] - The act of following a relationship from its source node to its target node.

type-ahead [R] -
The ability of a producer to transmit data units before the consumer requests the data units.

typing [R] -
An organization of entities, relationships and attributes in which they are partitioned into sets, called entity types, relationship

types and attribute types, according to designated type definitions.

uninterpreted data [R] -
A data representation whose structure is not controlled by CAIS facilities and whose structure is not used in the CAIS operations. Examples might be representations of files, such as requirements documents, program source code, and program object code.

unique primary path [C] -
The path from the system-level node to a given node traversing only primary relationships. Every node that is not unobtainable has a unique primary path.

unique primary pathname [C] -
The pathname associated with the unique primary path.

unnamed item [C] -
No name is associated with a list item.

unnamed list [C] -
A list whose items are all unnamed.

unobtainable [C] -
A node is unobtainable if it is not the target of any primary relationship.

user [C] -
An individual, project, or other organizational entity. In the CAIS it is associated with a top-level node.

user name [C] -
The key of a primary relationship of the predefined relation USER.

virtual terminal -
(Davies) A conceptual terminal which is defined as a standard for the purpose of uniform handling of a variety of actual terminals.

REFERENCES

DACS - DACS Glossary, a Bibliography of Software Engineering Terms, GLOS-1, October 1979, Data & Analysis Center for Software.

Davies - Davies, D. W., Barber, D. L. A., Price, W. L., and Solomionides, C. M., "Computer Networks and Their Protocols," John Wiley & Sons, New York, 1979.

IEEE - IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 729-1983.

K/K - KITIA Public Report, Volume I, 1 April 1982.

LRM - Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, Feb. 17, 1983, United States Department of Defense.

STONEMAN - Requirements for Ada Programming Support Environments "STONEMAN", Department of Defense, Feb. 1980.

TCSEC - Department of Defense Trusted Computer System Evaluation Criteria, Department of Defense Computer Security Center, CSC-STD-001-83, 15 August 1983.

UK Ada Study - United Kingdom Ada Study Final Technical Report, Volume I, London, Department of Industry, 1981.

WEBS - Webster's New Collegiate Dictionary, G. & G. Merriam Company, Springfield, Massachusetts, 1979.

@

An Investigation of the Common APSE Interface Set (CAIS) on an IBM S/370 Running VM/CMS

Jeffrey B. Vermetta

July 22, 1986

Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

INTRODUCTION

The Ada programming language was designed to facilitate the use of modularity, information hiding, exception handling, parallel processing, abstraction and other software engineering concepts by providing direct support to these techniques throughout the entire application development process. In support of this goal, the Ada language was intended to be one part of a support environment. In order for the Ada Programming Support Environment (APSE) to provide a rich set of tools which are both extensive and cost-effective, there must be a mechanism whereby tools can be easily ported among host development systems.

The Common APSE Interface Set (CAIS) provides specifications for a set of Ada packages which are designed to promote interoperability and transportability of Ada software development tools from one host development environment to another. The CAIS document defines interoperability as the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion. Transportability is defined as the ability of the tool to be installed on a different Kernal APSE (KAPSE) with the same degree of functionality in both systems.

The purpose of this investigation is to evaluate an implementation of a subset of the CAIS interfaces, several small CAIS tools and a command line interpreter (CLI). In order to accomplish this goal, the investigation can be broken down into the following four phases:

- Define a subset of the CAIS specifications which is robust enough to demonstrate most of the features of the CAIS.

- Implement the CAIS subset on the Virtual Machine / Conversational Monitor System (VM/CMS) operating system running on an IBM S/370 computer.

- Develop several small CAIS-like tools to aid in the evaluation of the CAIS packages.

- Develop a command line interpreter, a tool whose function is to interactively allow the user to access the CAIS interfaces and invoke the CAIS tools.

The CAIS subset can then be evaluated with respect to each of the different areas of interest which will be elaborated in the approach section. There are aspects of the CAIS which will not be included in the scope of this investigation, but which were taken into account during the design and development of the CAIS subset implementation. Thus the capabilities required to enhance this investigation can easily be incorporated into the present implementation. These areas are summarized as follows:

- Implement the CAIS subset on the Multiple Virtual Machine / Time Sharing Option (MVS/TSO) operating system running on an IBM S/370 computer.

- Enhance the functionality of the command line interpreter to support multiple users simultaneously.

- Port the tools written for the VM/CMS CAIS system to the MVS/TSO CAIS system.

These enhancements would allow several other aspects of the CAIS interfaces to be evaluated. The degree to which an implementation of the CAIS depends on the host operating system can be determined when porting the VM/CMS version of the CAIS to MVS/TSO. The issues of access control and security can be examinad when supporting several users on the same CAIS database at the same time. The degree to which the CAIS fulfills the objective of making tools portable can be evaluated by examining the difficulties and amount of changes needed to port the CAIS tools from the VM/CMS system to the MVS/TSO system.

APPROACH

The proposed CAIS policies explain that the principal purpose of the current release of the CAIS specification is to allow implementors to evaluate the CAIS interfaces as one component of an Ada programming support environment. This investigation was designed to evaluate the CAIS in each of the following three areas:

- An effort was made to implement a subset of the CAIS interfaces. This was done to demonstrate the capability of the given interfaces to provide a level of functionality sufficient to accomplish the objectives of the CAIS.
- Tools were written which run on top of a CAIS implementation and depend on the CAIS interfaces exclusively; that is, the tools function without regard to the underlying operating system. These were used to evaluate the usablity of the CAIS interfaces from the point of view of a tool writer.
- A command line interpreter was developed to allow the user to access the CAIS interfaces and invoke the CAIS tools interactively. In this manner, the CAIS subset, together with the CAIS tools which were written for the investigation, was examined and evaluated while running as a complete software development environment.

These three areas provided a means to evaluate the CAIS interfaces with respect to a number of different criteria:

- Performance - The response time necessary for a particular operation should not be significantly increased by the overhead necessary for the CAIS to maintain the node model.
- Efficiency - The CAIS implementation should make an optimal balance between the time and space resources involved.
- Usability - The CAIS should be both natural and understandable in its interfaces to the user and the CAIS tools.
- Viability - It should be a fairly straightforward process to add functionality to the CAIS system as new capabilities are desired.
- Feasibility - The CAIS interfaces should be a reasonably simple and natural approach to representing APSE entities.
- Reliability - Error conditions which occur in either the CAIS interfaces or any CAIS tools should be handled in such a way as to prevent any exception from compromising the data integrity of the node model.
- Completeness - The system should not require any resources or interfaces which are not available from the CAIS specifications.

CAIS SUBSET DEFINITION

This section describes the issues and difficulties found in selecting the interfaces which were included in the CAIS subset and the rationale for the decisions made during this process.

CAIS INTERFACES SUPPORTED

The subset was defined with the intention of supporting most of the features of the CAIS. This would enable the evaluation to work on a realistic subset; that is, the subset must be robust enough to accurately reflect a typical CAIS environment.

The hierarchical node model representing APSE entities consisting of file, structural and process nodes is fully supported. Linkages between nodes in the form of primary and secondary relations are also fully supported. The nodes, together with the relationships that link them, form the basis for the CAIS representation of APSE entities.

Attributes are supported for both nodes and relationships in the CAIS subset to describe the characteristics of entities in the node model. These interfaces were considered highly important, as they would undoubtedly be used by most CAIS tools in an actual CAIS environment.

The CAIS specification includes support for page, scroll and form terminals. However, only the form_terminal package was included in the CAIS subset, since one kind of terminal support would be sufficient for the investigation. Form terminal support was chosen because the development of the subset was performed on IBM 3270 series terminals, which are form terminals.

Single-volume labeled and unlabeled magnetic tape files are supported by the magnetic_tape package of the CAIS specification. The package was designed to conform to level 2 of the ANSI 78 standard. This package was considered important, but not necessary, to the investigation of the CAIS, because it demonstrates the ability of the CAIS to support a specific type of device in accordance with a separate standard. This is also important in that it shows how future enhancements may be easily made to the CAIS to support more devices and capabilities.

CAIS INTERFACES NOT SUPPORTED

Some of the features of the CAIS were not included in the implementation, as they were not considered necessary to the scope of this investigation.

To enforce security in a CAIS environment, the CAIS provides a set of interfaces to support mandatory and discretionary access controls. Mandatory access control is defined in the CAIS as being a set of controls based directly on a comparison of the individual's clearance or authorization for the information and the classification or sensitivity designation of the information being sought. Discretionary access control is a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. Although the capability is present in the chosen subset to support discretionary access controls through the use of node and relation attributes, these attributes are not predefined and enforced by the implementation.

Invocation of processes is not supported on the VM/CMS implementation of the CAIS subset because of the single-process nature of VM/CMS. However, process nodes are supported by the implementation and full process support could be added in the future on an MVS/TSO implementation. Thus, although processes are not actually spawned by the CAIS implementation, this activity can be simulated and the dummy processes can be tracked through the use of process nodes.

The text_io, direct_io, and sequential_io input/output packages defined in the CAIS specification are not supported. Instead of implementing the CAIS input/output packages, a new interface was defined to implement only those procedures which differ from the predefined Ada packages.

Support for page and scroll terminals was excluded in favor of form terminals, since IBM 3270 series terminals were used during the investigation, and a single type of terminal support was sufficient for the scope of this investigation.

INTERFACES ADDED TO THE CAIS SUBSET

In some cases, interfaces were defined which were not in direct support of the CAIS specification, in order to deal with difficulties in implementing the CAIS interfaces.

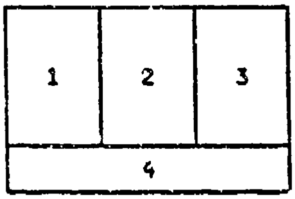
An Ada package called file_nodes was added to interface to text_io, direct_io, and sequential_io, the predefined Ada input/output packages. This package provides support to create file nodes, similar to the structural_nodes package defined in the CAIS specification. The package implements all of the input/output procedures which require parameters of type node_type, since these procedures differ from the predefined Ada packages. This allows the CAIS to utilize the predefined Ada packages directly.

An Ada package called cais_additional_utilities was added to define all functions and procedures required by this implementation that were not supported by the CAIS specification. These include support for backing up the CAIS database to a set of disk files and restoring them to resident memory, and any associated functions and procedures which were necessary to the implementation of the CAIS subset, but which were not included in the CAIS specification.

CAIS COMMAND LINE INTERPRETER

The Command Line Interpreter is a set of utilities controlled by a small driver routine which allow the user to interactively access the CAIS interfaces and invoke the CAIS tools. Over the course of this investigation, the CLI changed and matured greatly. At first, the CLI simply allowed the user to type in simple commands to create and delete structural nodes. Additional capabilities were added over the course of the investigation until the CLI was a powerful tool capable of supporting all of the CAIS interfaces as well as invoking the various CAIS tools.

The CLI became a full-screen interactive CAIS environment driver. The user interface was supported by both a command line and a complete menu-driven interface and was divided into four areas as shown in the following diagram:



AREA 1: On the left part of the screen, items could be selected from a set of menus listing all of the CAIS operations and CAIS tools. Menus could be scrolled forwards and backwards with function keys.

AREA 2: In the center third of the screen various parameters describing the state of the CAIS environment were displayed. These included the current time and date, the total number of nodes in the system, the number of nodes owned by the user, the number of users in the system, the number of operations which had occurred since the last time the system was backed up and the time of the next scheduled backup.

AREA 3: The right part of the screen displayed the kinds, relation names and relationship keys of the nodes representing the current user, the current node under that user, and the children described by primary links originating from the current node.

AREA 4: The bottom of the screen was used as a command line where the user could type in requests directly rather than use the menus. This area was also used whenever the CLI prompted the user for an input string, such as a file name or a relationship key.

A separate screen could be activated to display all of the information about the current node including the names, types and values of the node and relation attributes, the name of the host file associated with the node's contents and the kind, relation name and relationship key of the node's parent.

By allowing the user to interactively access the CAIS interfaces, the command line interpreter provided a means for the CAIS database to be built and tested before tools were available. This greatly facilitated the debugging process of the interfaces, since many different operations could be attempted and their results on the node model could be examined. The implementation of the CAIS interfaces would have progressed much more slowly had it been necessary to write small driver routines to test each of the procedures as they were developed.

CAIS TOOLS

As the CAIS subset matured to the point where file nodes and the import and export procedures were supported, it became possible to begin testing tools in the CAIS environment. Several small tools were developed, including an Ada source line counter and a pretty printer. Initially, these tools could only be invoked from within the command line interpreter. As the CLI developed more capabilities, more tools were integrated, including a stubber, a more powerful pretty printer, a compiler batch submissions tool and support for two full-screen editors. Support was added in the command line interpreter to allow the user to invoke these tools interactively, with several different levels of functionality. These different levels were designed to give the tools more and more responsibility to access the node model, thereby allowing the tools to be tested and developed with successive increases in functionality. These classes of interfaces are as follows:

LEVEL 1: At the minimal level of CAIS functionality, a given tool would work directly with a host file. The command line interpreter would export the file from the current node, and that file would be passed as input to the tool. This allowed non-CAIS tools to be used at this stage of investigation, since the tools had no access to the CAIS interfaces.

LEVEL 2: The current node was opened by the command line interpreter and passed to the CAIS tool as input. The tool was responsible only for extracting the host file from the node's contents and had no direct contact with the node model. The results of the operation performed by the tool were put into a file which the tool then passed back to the CLI. The CLI would then create a new node as a child of the current node and import the results file to the new node.

LEVEL 3: The command line interpreter passed the current node to the tool, which would extract the file, perform its operations, create a results file, create a new node and import the results file to the new node. This is the first level at which the tool itself changed the CAIS database.

LEVEL 4: At the final level of tool development, the command line interpreter would simply pass control over to the tool when the user invoked it. The tool would be responsible for all interaction with the CAIS database. When the tool was finished, it would pass control back to the CLI. This level of responsibility is equivalent to a tool running by itself without a command line interpreter.

The method of iterative development of tools proved extremely informative in that the difficulties of writing tools could be examined in several stages. This provides a much more thorough examination of the usability of the CAIS interfaces from the point of view of the tool writer than if the tools had simply been developed in one pass and evaluated after they were completed. Integrating tools in an iterative fashion displayed very clearly the tradeoffs between the level of functionality of the tools and the level of difficulty in implementing the tools. This could be considered to be a measure of the usability of the CAIS interfaces, which is an extremely subjective characteristic and hence difficult to evaluate.

EVALUATION

The entire process of defining a subset, implementing the subset, developing several CAIS tools and developing a command line interpreter proved extremely useful in evaluating the CAIS specification. One of the most important aspects of the investigation was the iterative approach which was used. This approach allowed the CAIS to be developed at the same time as the command line interpreter and the tools, in a series of iterations which provided input to the evaluation at each step of the investigation.

At the completion of the investigation, there was a total of over sixteen thousand lines of Ada source code running in the CAIS environment. Lines of code, for the purposes of this measurement, included fewer than five percent comments and blank lines and a single Ada statement which ran over several records would be counted as several lines of code. The code can be broken up into several basic categories to gain a better understanding of the size of the investigation:

CAIS Interfaces Command Line Interpreter CAIS Tools Terminal Interface	3,164 lines 2,854 lines 5,137 lines 4,946 lines
Total	16,101 lines of code

The evaluation of this investigation was broken up into three areas: the CAIS subset, the command line interpreter and the CAIS tools.

CAIS SUBSET

The two basic criteria that were used in selecting a subset of the CAIS interfaces to be implemented for this investigation were that the subset defined should be robust enough to demonstrate most of the features of the CAIS and simple enough to be implemented in a reasonably short amount of time. Some tradeoffs were necessary to achieve a reasonable balance between these two objectives. For example, only one of the three terminal support packages was chosen to be included in the subset, since support of one type of terminal would be sufficient to demonstrate the ability of the CAIS interfaces to support terminals.

Initially, an extremely severe subset was defined and implemented. This subset included only those interfaces necessary for the creation and deletion of structural nodes and for defining primary links between the nodes. This allowed the testing of various data structures to represent nodes and relationships, without requiring a large amount of programming when changing data structures. After the implementation-defined data structures were finalized, the iteration process began whereby the CAIS interfaces and the command line interpreter would be developed in parallel in order to facilitate the testing and integration of the interfaces. When the CAIS interfaces had matured to the point of supporting file nodes and their manipulation, several simple CAIS tools were defined and their development was included in the iterative process.

By employing successive increments in the level of functionality of the CAIS interfaces and including thorough testing through the use of the command line interpreter as a driver for the interfaces, the original subset was gradually enhanced until it reached its final form. In the order in which the interfaces were added, the final subset of the CAIS interfaces used in this investigation included structural nodes, primary links, file nodes, file import and export operations, secondary links, full node management support, form terminal support, process nodes, list utilities, node attributes, Ada input/output support, relation attributes, backup and restoration of the CAIS system and magnetic tape support.

The final subset included most of the features of the CAIS, but several were deliberately excluded from the scope of the investigation. These interfaces include those for security, process control and input/output operations.

Security was not included in the subset because the scope of the investigation was limited to a single-user environment. Most of the discretionary access controls are much more appropriately addressed in a multi-user environment. The attribute operations provide a set of interfaces which can easily be used for access control, however, and the attribute operations are fully supported by the final subset.

Process nodes can be created in the final subset, although real processes are not actually spawned as part of the creation operation. Through the use of attributes, all of the pertinent characteristics of processes can be maintained, such as the start and stop times. The reason the processes are not actually spawned, but merely simulated, is that the VM/CMS operating system which this CAIS implementation runs on does not support multiple processes running on a single virtual machine. Thus, only the tracking and simulation of processes is included in the CAIS subset.

Instead of supporting the three input/output packages defined in the CAIS specification, the subset defined a new interface which imported the three predefined Ada packages. This decision was not made because the CAIS interfaces were considered inappropriate, but because of the time restraints imposed on the development of the CAIS subset. A small package could easily be defined to extract the host file from the file node and allow the predefined Ada packages to operate on the host file, without performing an export operation. In this manner, a tool would operate just as if it were performing input and output operations on the contents of a node, but would actually be working with the host file represented by the contents of the node. This was considered to be an acceptable compromise, allowing support of the input and output functionality described by the CAIS specification without requiring the actual implementation of these three packages.

In order to maintain a file-based copy of the CAIS system, several interfaces were added to the subset to support the backing up and restoration of the nodes and relationships. These procedures would save and restore all of the information contained in the data structures which represented the node model in several host files using direct and sequential input/output operations.

CAIS COMMAND LINE INTERPRETER

The command line interpreter provided a means to develop a very usable application programming environment based on the CAIS interfaces. This provided a great deal of valuable input about the usability, performance, efficiency, viability and reliability of the CAIS interfaces from the point of view of the user. The CLI proved to be an extremely flexible and powerful tool for running a CAIS environment. For approximately the last two months of this investigation, when it had matured enough to make usage possible, the CAIS environment was driven by the CLI and actually used as the development environment for this project.

In order to use the CAIS as a true development environment, it was necessary to integrate two non-Ada tools, an Ada compiler batch submission tool and a full-screen editor, into the CAIS system. Although these tools were not developed into CAIS tools, they were considered valuable to the investigation because they allowed the CAIS to be used as a working environment, rather than just an experimental set of interfaces.

All of the other tools and many of the non-critical CAIS interfaces were developed, tested and integrated from within the running CAIS environment. The CLI allowed the user to edit source programs, print hard copies, compile Ada code, test new routines, and integrate them into the running environment. The command line interpreter itself was improved and modified from within the CAIS environment. The development of these tools and interfaces showed that the CAIS interfaces can be used to define an application programming environment which is very usable from the user's point of view.

It is a subjective effort to try to evaluate the usability of the CAIS, so no qualitative measurement will be attempted, but it can be said that, in general, the CAIS environment was no more difficult to understand and use than a typical operating system. Of course, the subset of the CAIS used in this investigation was not nearly as powerful as a typical operating system, nor did it provide many of the functions of an operating system, but it did prove to be extremely fast and reliable.

The performance and efficiency of the CLI proved to be much higher than anticipated. For nearly all of the operations which involved the CAIS interfaces, the response time was immediate. CAIS operations were performed as fast as they could be typed. The reason for this performance lies in the fact that the CAIS database was maintained both in host files and in data structures. All CAIS operations were carried out on the data structures in memory and only when the system was backed up did these changes get reflected in the host files.

The issue of reliability was resolved by maintaining the file-based backup copy of the CAIS database. The backup and restore procedures each took about one minute to execute, but this was considered to be a reasonable tradeoff for the increase in reliability gained by having a file-based copy of the database. Instead of losing the entire CAIS database if the computer suffered a power failure, for example, all that would be lost are those operations which occurred after the last time the system was backed up. The CLI would automatically backup the database at regular intervals. Also, after any particularly important operation, the user could request that the system be backed up immediately. In this manner, very few CAIS operations would ever be lost, so the CAIS environment had a high degree of reliability.

In one respect, this method of maintaining two copies of the CAIS database was even more reliable than a purely file-based CAIS. In a file-based system it would be difficult to recover if for any reason the data integrity of the CAIS database was compromised. In the two-copy environment, however, all that needs to be done if the database is compromised is to restore the most recent backup of the database.

The command line interpreter proved to be extremely valuable in the evaluation of the CAIS. By allowing the user to interactively invoke the CAIS interfaces and tools, it assisted greatly in the testing and integration of the interfaces. In addition, it permitted the CAIS environment to be used as the application programming environment for much of the investigation, thereby gaining much insight into the areas of performance and reliability. Overall, the command line interpreter can be considered a very natural and powerful tool in the development and testing of a CAIS environment.

CAIS TOOLS

The tools developed for this project were deliberately chosen to represent a wide range of capabilities and complexity. Ranging from the simplest to the most complex, they included an Ada source line counter, a basic pretty printer, a body stubber, an Ada compiler batch submissions tool, a complex pretty printer and two full-screen editors.

The change in size of each of the tools over the course of the investigation is shown in the following table, broken down into the four levels of responsibility. As an example, the simplest tool was the Ada source code line counter, which read in a source file and produced a report showing the number of lines of Ada code, comments, semicolons, blank lines and total lines in the file. The line counter itself contained only 85 lines of code at the first level of CAIS responsibility discussed earlier. This was the level at which the tool had no direct interaction with the CAIS interfaces. At the second level, with responsibility for read accesses to the CAIS, the size of the line counter increased to 120 lines of code. Adding the third level of responsibility increased the line counter to 185 lines of code. At the fourth and final level, where the tool is fully responsible for all interaction with the CAIS database, the line counter reached its maximum length at 225 lines of code.

Level	Line Counter	Pretty Printer	Body Stubber	Batch Submit	Pretty Printer	Screen Editors
First	86	887	852	N/A	2430	N/A
Second	120	930	899	XXXX	2491	XXXX
Third	182	1007	992	XXXX	XXXX	XXXX
Fourth	225	1039	XXXX	XXXX	XXXX	XXXX

The batch submissions tool and the two full-screen editors were not written in Ada, so their line counts were not included. They were included in the CAIS environment because the CAIS system was actually used as a programming environment for much of the investigation. Since full-screen editing and Ada compiling were both necessary for any application programming environment, these tools were necessary, even though they were never developed beyond the first level.

The rest of the tools that were developed for this project were deliberately chosen to form a representative set of a large class of tools; specifically, tools which read in data from a host file, perform some function on that data, create a second host file and write the results of the function into the second host file. The line counter is a very simple example of such a tool, while the pretty printer is a much more complex tool of this same type. The reason for this choice becomes apparent when the attempt is made to measure the usability of the CAIS interfaces.

Usability is a very subjective concept, and an objective method to measure the usability of the CAIS interfaces from the point of view of the tool writer is difficult to specify. One aspect of usability which can be measured, however, is the degree of localization of the changes required to convert a non-CAIS tool into a CAIS tool. By modifying the tools in four steps as described above, the size of each tool was tracked throughout the course of the investigation. A measure of the usability of the CAIS interfaces can then be defined as the tendency of the increase in size of each tool to be constant regardless of the size of the tool, as each tool is developed in the four steps. This is because a very usable interface would not require changes in the structure or logic of a tool, but only localized changes to such areas as input/output and exception handling. Such changes would involve around the same number of modifications regardless of the original size of the tool. In short, then, if each tool grew by an amount proportional to the size of the tool, it would indicate a poor degree of usability. On the other hand, if each tool grew by a relatively constant number of lines, it would indicate a high degree of usability.

It is apparent from the table that the amount of change required to give the tools more responsibility was relatively constant regardless of the original size of the tool. For example, the line counter increased by 139 lines and the first pretty printer increased by 152 lines, even though the pretty printer started out over ten times the size of the line counter. The changes to the tools in each case occurred in those areas which dealt with initialization, input/output and exception handling.

This suggests that many non-CAIS tools could probably be developed into CAIS tools in a fairly straightforward manner by changing the initialization, I/O and exception handling routines. The rest of the tool can be left virtually unchanged. This is valid only for tools which are similar to those used for this investigation, however. A tool which monitors machine usage, for example, would not fall under this category.

It should also be noted that it takes less time and effort to convert additional non-CAIS tools into CAIS tools, since many of the changes are very similar to those made during the conversion of earlier tools. For example, it took significantly less time to convert the body stubber into a CAIS tool than the line counter, because much of the work was copied from the line counter into the body stubber with little or no modification. This is another indication of the high degree of usability of the CAIS interfaces.

Since production-quality tools designed for the CAIS are not yet available, a likely alternative for implementors is to take existing tools and convert them into CAIS tools. The four successive levels of responsibility used in this investigation proved to be very helpful in developing these tools, since they could be tested at each level, with only minor changes occurring at a time. The CAIS interfaces are very usable from this standpoint, since very similar changes were made to each tool to progress from level to level. Overall, then, the CAIS interfaces had a high degree of usability, as seen in the objective measurement of the degree of localization of changes necessary for the conversion of a tool.

SUMMARY

Throughout the course of this investigation, particular attention was paid to the evaluation of the CAIS interfaces from the points of view of the CAIS implementer, the CAIS tool writer and the user of the CAIS environment. The design and development of the CAIS subset, the command line interpreter and the CAIS tools all were influenced by this objective.

The CAIS interfaces proved to be fairly straightforward and understandable during the implementation of the subset. The node model seemed to be a very natural way to represent the APSE entities, and lent itself to several different data structures in a fairly clear and understandable way. This greatly aided the implementation of the CAIS interfaces during the investigation.

From the perspective of the tool writer, the CAIS proved usable in that it did not impose any particular restrictions or requirements on the structure and logic of the tool. Rather, most of the interactions with the CAIS interfaces occurred in several localized areas of the tools. This greatly reduced the degree to which the CAIS interfaces had to be considered when designing and developing the tools. This also allowed non-CAIS tools to be converted into CAIS tools in a straightforward and fairly simple manner.

Finally, the use of the CAIS system as an actual application programming environment demonstrated the excellent response time performance, the usability of the node model and the reliability the CAIS interfaces are capable of providing to the user of the CAIS environment.

BIBLIOGRAPHY

Barnes, J.G.P., Programming In Ada,
Addison-Wesley Publishing Company, 1982.

Booch, Grady, Software Engineering With Ada,
Benjamin/Cummings Publishing Company, Inc., 1983.

Military Standard, Common APSE Interface Set,
Department of Defense, Jan 31, 1985.

Referance Manual for the Ada Programming Language,
United States Government, Feb 17, 1983.

Requirements for Ada Programming Support Environments,
Department of Defense, Feb, 1980.

An Investigation of the Common APSE Interface Set On an IBM S/370 Running VM/CMS

Jeffrey B. Vermette
IBM Federal Systems Division

07/21/86

Unclassified

- The scope of this investigation was to implement a CAIS environment sufficient to evaluate the CAIS with respect to three points of view:

  - CAIS Implementor

  - CAIS Tool Writer

  - CAIS Environment User

Major Areas Of Investigation

- The investigation can be broken down into three major areas, each of which supports one of the three points of view mentioned above:
  - CAIS Interfaces
  - CAIS Tools
  - Command Line Interpreter

Criteria for Evaluation

- The investigation attempted to evaluate the CAIS in each of the following criteria:
    - Performance / Efficiency
    - Usability
    - Viability
    - Feasibility
    - Reliability
    - Completeness
    - Ease of Implementation

- The CAIS Subset implemented in this investigation was chosen with two basic criteria:

  - Subset should be robust enough to demonstrate most of the features of the CAIS and hence be a reasonable representation of a CAIS environment.

  - Subset should be simple enough to be implemented in a reasonably short amount of time.

- Some tradeoffs occurred between these two criteria in reaching the final CAIS subset.

CAIS Interfaces Supported IBM

- The CAIS Subset implemented in this investigation included the following CAIS Specification interfaces:

  - Node Definitions

  - Node Management

  - Node Attributes

  - Relaticn Attributes

  - List Utilities

  - Form Terminal

  - Magnetic Tape

- The CAIS Subset implemented in this investigation did not include the following CAIS Specification interfaces:
  - Mandatory and Discretionary Access Control
  - Process Spawning
  - Input/Output packages
  - Page and Scroll Terminals

- The following interfaces were added to or replaced interfaces of the CAIS Specification:

  - Package FILE_NODES

  - Package CAIS_ADDITIONAL_UTILITIES

- The tools used in the investigation covered a wide range of size and functionality.
  - Line Counter - developed for this investigation
  - Pretty Printer 1 - completely integrated
  - Body Stubber - completely integrated
  - Pretty Printer 2 - partially integrated
  - Batch Compiler Submitter - partially integrated
  - XEDIT Editor - partially integrated
  - SPF Editor - partially integrated

- Tools were integrated into the CAIS system in four stages in order to more easily evaluate the usability of the CAIS interfaces from the point of view of the tool writer.
    - Level 1: Tool has no responsibility for the Node Model. (No Access)
    - Level 2: Tool has responsibility of extracting CMS file given the appropriate file node, but not of changing the node model. (Read Only)
    - Level 3: Tool has responsibility of extracting CMS file given the appropriate file node and of creating any new nodes as the result of the tools operation. (Read/Write given current node)
    - Level 4: Tool responsible for all interactions with the node model (Read/Write)

- The command line interpreter was developed for this investigation as a full-screen, menu-driven user interface to the CAIS interfaces and tools. The CLI screen was composed of four sections:
  - Menu section - contained menus of available options.
  - Status section - contained information about the current state of the CAIS system.
  - Display section - contained list of current user, current node, and primary relationships from current node.
  - Command Line - used for user input of commands and information needed to process menu requests.

- A separate screen could be activated to display all of the information regarding a node, including attributes and associated CMS file.

- The iterative development approach taken in this investigation greatly aided in the areas of testing and evaluation.

  - The CAIS Subset could be implemented and tested interactively from within the command line interpreter, rather than with short test programs and a dummy CAIS database.

  - Relative complexity of CAIS interfaces could be more readily understood.

  - Tools could be integrated into the CAIS environment much more easily by defining levels of responsibility for the tools.

  - The CAIS system itself could be used as the development environment of the investigation.

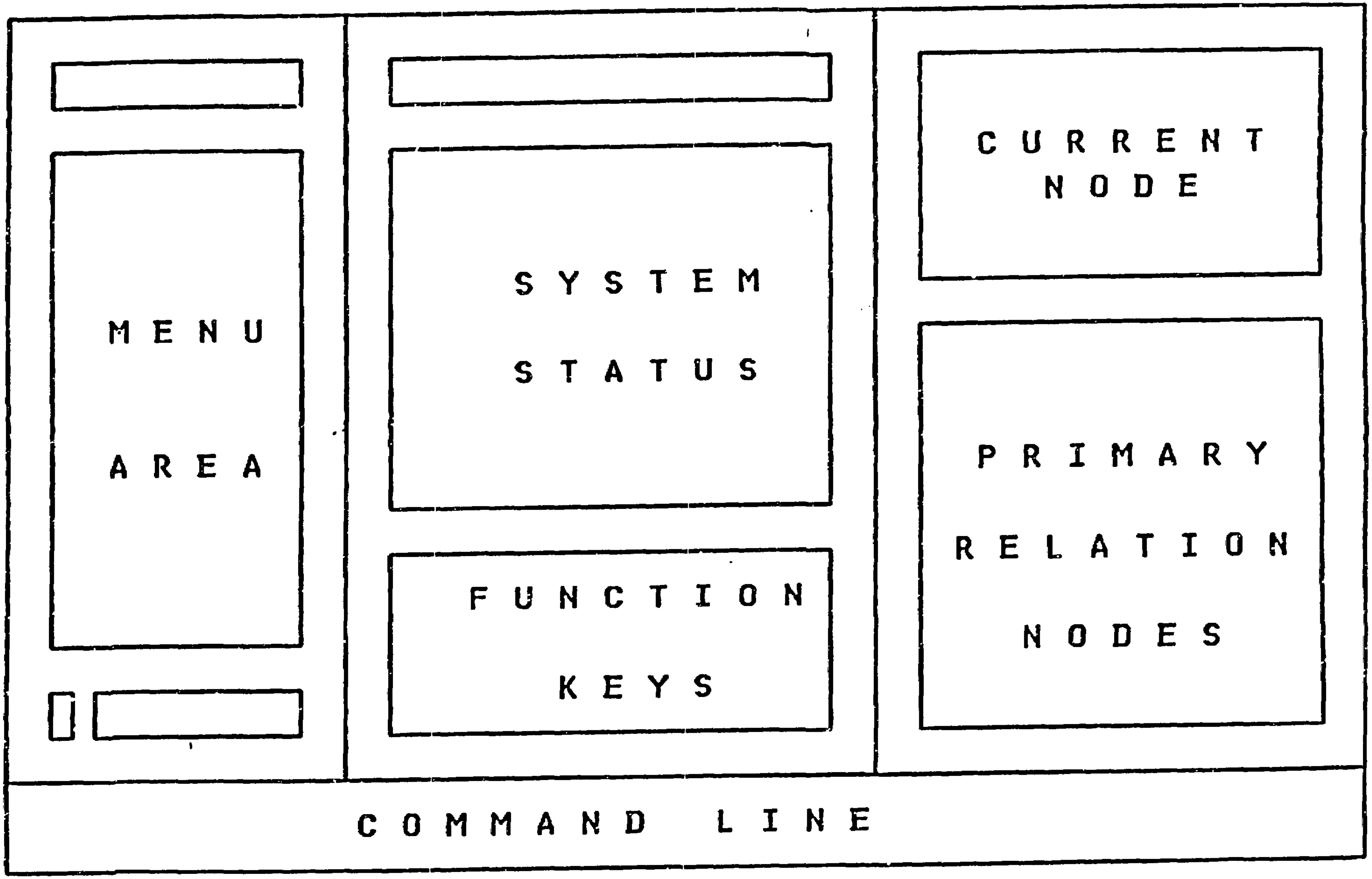
- The CAIS Subset was initially defined to be extremely severe in order to examine the performance of several data structures.

- After finalizing the data structures of the node model, the subset was successively enhanced in parallel with the command line interpreter and the CAIS tools.

- The final subset was robust enough to demonstrate most of the features of the CAIS.

CAIS Investigation Code Size

Description	Lines Of Code
CAIS Interfaces	3,164
CAIS Tools	2,854
Command Line Interp.	5,137
Terminal Interface	4,946
TOTAL	16,101


MENU
AREA
SYSTEM
STATUS
FUNCTION
KEYS
CURRENT
NODE
PRIMARY
RELATION
NODES
COMMAND LINE

3-598

LEVEL	LINE COUNTER	PRETTY PRINT 1	BODY STUBBER	BATCH COMPILER	PRETTY PRINT 2	SCREEN EDITORS
1	86	887	852	N/A	2430	N/A
2	120	930	899	---	2491	---
3	182	1007	992	---	---	---
4	225	1039	---	---	---	---

Submitted to Ada Letters

Implementation of a prototype CAIS environment.

by

P. Carr, R. Stevenson, J. Alea, J. Berthold, G. Croucher, M. Davis,
G. Dobbins, D. Law, V. Szarek and W. Webster.

Gould Inc., Computer Systems Division,
6901, W. Sunrise Blvd.,
Ft. Lauderdale, Fl, 33313-4499, USA.

1 July, 1986

Abstract

This paper describes a project to investigate the feasibility, performance and utility of a CAIS compliant Ada Programming Support Environment. A working model of an environment was built, with a command language interpreter and a small toolset. Tools from the host environment have been imported and made to behave as native CAIS tools. A number of tools have been ported from a parallel effort by a MITRE corporation team with little difficulty. The prototype was built initially for correctness and enhanced later for performance improvements. The performance was found to be acceptable as a test bed for development of prototype tools and offered hope for the performance of a product-quality implementation. A test suite for compliance with the standard was partially implemented.

1.0 Introduction.

The Common APSE Interface Set (CAIS) is designed to provide a common, host-independent model for Ada[1] programming development tools and toolsets. It is currently specified in a Proposed Military Standard [1] with a rationale document [2] as a commentary about the authors' intentions. A Reader's Guide [3] gives a more readable summary of the CAIS. These documents have been produced by a group called the KAPSE Interface Team (KIT) with support from Industry and Academia (KITIA). The proposed standard is intended to be the first stage towards a more ambitious and far-ranging standard covering aspects such as distributed environments, inter-tool interfaces, graphical interfaces, etc. It is intended that the second version should be a compatible superset of the current version.

This paper describes a prototype of a CAIS environment, implemented on a host operating system and designed for correctness rather than efficiency. Some performance figures are given as a guide for future designers as to the likely performance of a usable, product-quality CAIS-compatible environment.

2.0 Overview of the CAIS.

The CAIS specification describes an operating environment which is independent of the underlying system. All objects in the system are mapped to nodes. Each node has a set of properties, or attributes, which can be read using CAIS interfaces. Some of the attributes have values which are properties of the node and cannot be changed by the user. Others have values which can only change when a status change occurs. The

[1] Ada is a registered trademark of the U.S. Department of Defense (AJPO).

attributes described so far are called predefined attributes. Other attributes have user-defined names and values and can be changed using CAIS interfaces.

The nodes are connected by relationships. Primary relationships join the nodes to form a strict tree, rooted in a system node. The removal of a primary relationship implies the removal of the node from the model. Secondary relationships can point to any node except the system node, which is inaccessible to the user.

Relationships are labelled by relation names, which can be predefined or user-defined in the same way as attributes. All relationships have keys. Multiple relationships with the same relation name are distinguished by keys. Relationships can have attributes which behave exactly in the same way as node attributes.

Nodes are used to describe three main kinds of objects. These are files, structural nodes and processes. File nodes consist of devices, storage files and queues. Queues are like pipes or mailboxes. Structural nodes serve as place-holders for relationships. A structural node which is the parent of file nodes only is the equivalent of a directory in a host system. The process node normally represents the execution of an Ada program which may consist of one or more Ada tasks. Processes may also be written in other languages.

The CAIS environment is perceived by APSE tools as a hierarchical file system with process trees whether the host system implements these features or not. The environment supports a number of interesting features such as a unified pathnaming scheme for all objects in the system and the ability to form mixed trees of processes and files. The intention is to facilitate production of toolsets consisting of several cooperating processes with flexible communication and simple facilities for temporary file and directory creation. For example, when a process node is deleted, its offspring must also be deleted. Files created below the process in the tree are automatically removed.

Pathnames are formed by concatenating the relation-key sequences to traverse intermediate nodes. All pathnames used in CAIS service calls are considered to start at the current process node. For the purposes of unique identification, a node is considered to have a primary pathname, which is the path from the system node to that node using primary relationships only. However, primary pathnames are never used to identify nodes to CAIS services.

The CAIS specification identifies a static structure of top-level nodes, including device nodes, user nodes and role nodes, which cannot be added to or removed from within the CAIS environment. Device nodes model the devices supported by the environment. User nodes are structural nodes that become the top of the user's local tree when the environment is entered. Role nodes define access control groups. The implementation can structure a top-level tree with any combination of file and structural nodes.

When a user enters the environment (by an implementation defined method), a user node is identified as the current user and a top-level process node is generated which is an offspring of the current user node via a primary Job relationship. This process node is a control program that defines the environment's appearance to the user. Normally, a command language interpreter would perform this function, but this need not be so. If the top-level process node exits or aborts, the user exits from the environment and the whole current process tree is terminated. From the top-level process node, other process nodes can be initiated in wait or no-wait mode as offspring processes. These processes, in turn, can initiate other processes. Any process can generate an independent process tree off the current user node or off any other accessible user node using the create_job service. The new process tree is not terminated when the current user logs out.

The CAIS provides node management facilities for general node and pathname manipulation. Other areas of functionality are:

1) a set of I/O packages modelled closely on the Ada packages to facilitate conversion of Ada tools to the CAIS standard.

2) an access control model based on current DoD concepts [4].

3) a virtual terminal interface modelled on industry standard terminal functionality.

4) a standard magnetic tape interface based on the ANSI standard [5].

5) an import-export package to transport files from and to the host system.

6) a standard list format based on a restricted form of the Ada aggregate format. This is used to pass complex information from the user to the CAIS and to pass parameters between processes.

7) a model of Queue handling designed to simplify the construction of Command Language Interpreters.

8) a process control model designed to be simple enough to be implemented on most systems currently in existence.

3.0 Goals of the prototype implementation.

It was felt that the CAIS needed working implementations because:-

a) the successive versions of the CAIS specification had received much criticism on the grounds of performance, unimplementability and unusability.

b) a working model would enable potential users to evaluate, train or plan for the future and would stimulate interest in the unique features of the CAIS.

c) we and other tool manufacturers would be able to generate working CAIS compliant tools in readiness for the release of production quality CAIS environments.

d) the access control model in the CAIS is complex and rich in function. There was a need for self-education by implementing a literally correct prototype before going on to implement an efficient analog.

The immediate goals of our prototype were:-

a) Correctness.

b) Portability. There was to be as little use of OS-unique features as possible. The prototype was to be coded in Ada.

c) Re-usability. It was hoped that large parts of the code would be independent of the underlying model.

d) Education. The project provided training, on the CAIS and in Ada programming.

e) Timeliness. A literal implementation of the model of the nodes and their access mechanisms would produce useful results far faster than attempts to design an efficient, quality product from the start.

Implementation of the Magnetic_tape and Form_terminal packages was postponed, since these were the least essential packages for our environment.

4.0 Prototype Implementation.

4.1 Design strategy.

The design was undertaken using the structured design methods of Constantine and Yourdan [6]. It was found that the production of structure charts could not proceed beyond the first few levels without a clear idea of how the node database was to be implemented. Having settled on a basic approach and layout, we were able to specify a package for access to the nodes. It was then possible to complete the structured design of the node management packages. As this proceeded, we were able to refine the node access package interface. We were then able to complete the structured design of the latter package.

4.2 Hardware and Software Environments.

The software was implemented on a Gould Powernode[2] 9080 which is a tightly-coupled dual processor, 32-bit superminicomputer with a performance rating of around 10 Mips. The software is capable of running on any of the Gould Powernode 60xx and 90xx series computers.

The design of the CAIS prototype was targeted to both Gould's UNIX[3] implementation, UTX/32[2], and to its proprietary Real-Time Operating System, MPX-32[2] . It was implemented using the ICC[4] Ada translator (version 3.1), an unvalidated translator from Ada to the "C" language. It was intended to port the code to Gould's validated compiler when this became available on the two target operating systems. The software was demonstrated on UTX/32 using the ICC 3.1 translator in April 1986. The port to the Gould compiler on UTX/32 followed the latter demonstration.

4.3 Implementation of the Nodes.

The original design consideration of OS independence forced us to map the nodes onto disk files. Each component of the node (relations, attributes and contents) is kept in a separate file. A further file was designated the node private status to hold the private description of the node and information about its open status. The use of separate files was designed to reduce contention when parts of the nodes were locked, since only one of the target hosts supported record locking.

The nodes were to be held in a single host OS directory in the first implementation and access to them was to be limited to processes logging in through the CAIS login facility. The node filename formed the unique node pointer which is needed to track the node during its lifetime. Relationships point to the node even when it is renamed within the CAIS.

4.4 Implementation of the Node Handles.

The Node handle, which is returned when a node is opened, cannot be implemented exactly as specified in the Proposed MIL-STD-CAIS. This is because the node type is limited private to a single package, Node_definitions. Its contents are not only hidden from the user but also from the rest of the CAIS packages. The option to move it to an enclosing CAIS package so that the type becomes visible to all sub-packages but not to the user packages was rejected because the ICC translator, at that time, did not support separate compilation. The size of the CAIS would have made program development too time-consuming. Instead an internal package was used to perform the unchecked conversion from node_type (an access type) to node record pointer. Functions were provided to access and manipulate the handles from within the CAIS.

4.5 Access to the Nodes.

The two main approaches to the implementation of node access were:-

a) to implement one or more servers to mediate access requests to the nodes so that transactions could be serialized. The server's error recovery could always guarantee the consistency of the node model.

[2] Powernode, UTX/32 and MPX-32 are trademarks of Gould Inc.

[3] UNIX is a trademark of AT&T Bell Laboratories.

[4] Irvine Compiler Corporation.

b) to use direct locking on the parts of the file nodes and to perform Ada or Host I/O directly on the node parts.

The lack of a common message passing mechanism for both Host OS's tended to point away from option (a). Also a common server could become a performance bottleneck in any simple implementation on a multi-processor system.

Option (b) was selected for implementation but the following problems were foreseen:-

1) Every tool in the environment would be forced to carry all the CAIS package code, a large overhead in disk and memory space which could only be alleviated by the use of shared libraries.

2) The possibility of deadlocked processes would be increased by having many tools locking parts of the nodes at random.

3) Exit processing had to be performed by an outside entity so that nodes could be closed and the database returned to normality when aborts occurred.

4) The method lacked robustness since there was no protected database manager with inherent recovery facilities. System crashes or process aborts could leave the database in an inconsistent state.

5) There was no protection against concurrent use of the same node handle by separate tasks in a single process. Serialized transactions would at least have reduced some of the damage possible if a tool used a node handle in such an erroneous fashion.

The following precautions against items 1 to 5 were taken:-

a) Locking of more than one part of a node by a single process was done according to strict code of practice rules designed to allow processes to back off and retry if they could not lock all parts required in one request.

b) All opens and closes of nodes are reported to a job server whose functions include closing nodes left open after an inadvertent exit or abort of a process. The job server is described later.

c) An extra file, the semaphore file, was added to process nodes to coordinate suspension, resumption, abortion and deletion without fear of becoming deadlocked. A semaphore package implemented functions for set, reset, test and test-and-set so that the semaphore functionality could be re-implemented in shared memory if this became available in the future.

d) Automatic node checking on the lines of UNIX file checking was to be implemented incrementally.

e) The prototype would ignore the dangers of user misuse of node handles described in item 5.

I/O to the nodes used Ada I/O of variant records to implement linked list structures for relation and attribute node parts.

So far deadlocking of processes and robustness have not been an issue. The node checking and repairing facility has been designed but only a subset has been implemented.

4.6 Access synchronization.

At a higher level than the locking of individual node parts, the CAIS requires that separate processes contend for access to nodes using a host-independent access synchronization model. Each process opening a node states for which intents the node is to be opened. There are 26 intents, based on combinations of exclusive or shared, read, write or append, and access to all or part of the node. When a requested intent is incompatible with currently open intents on the node, the

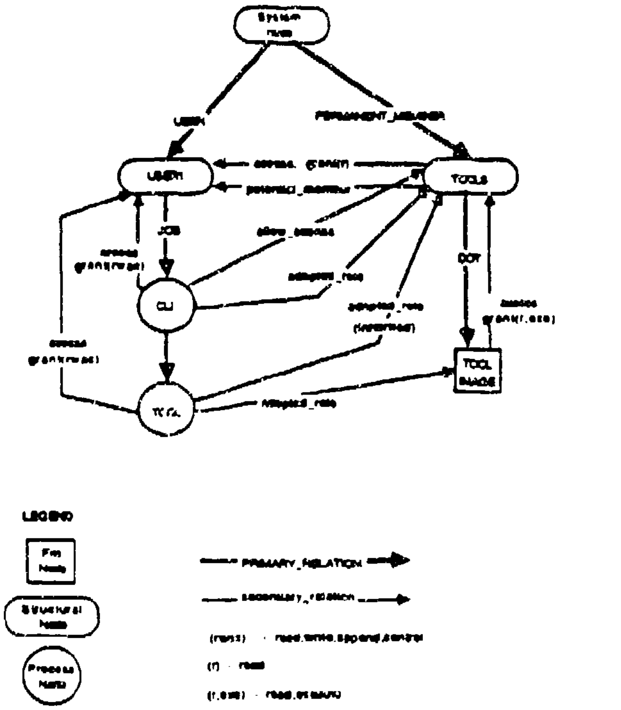


Figure 1. CAIS Access control relationships.

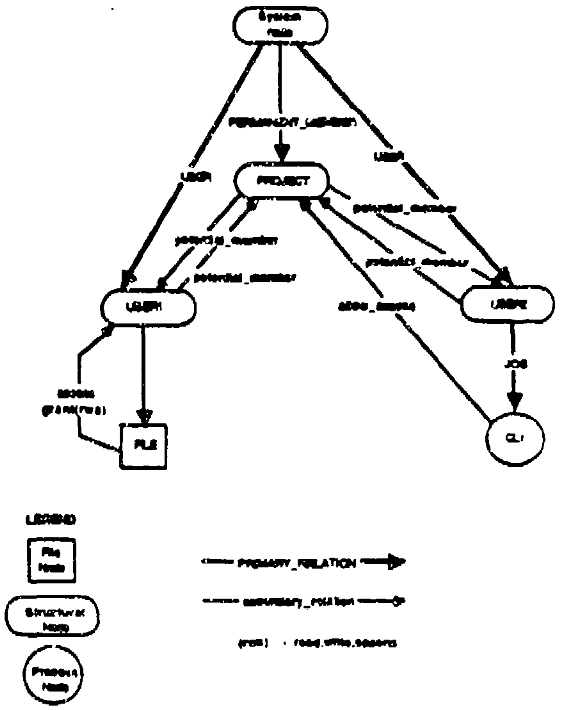


Figure 2. Access to other user node trees.

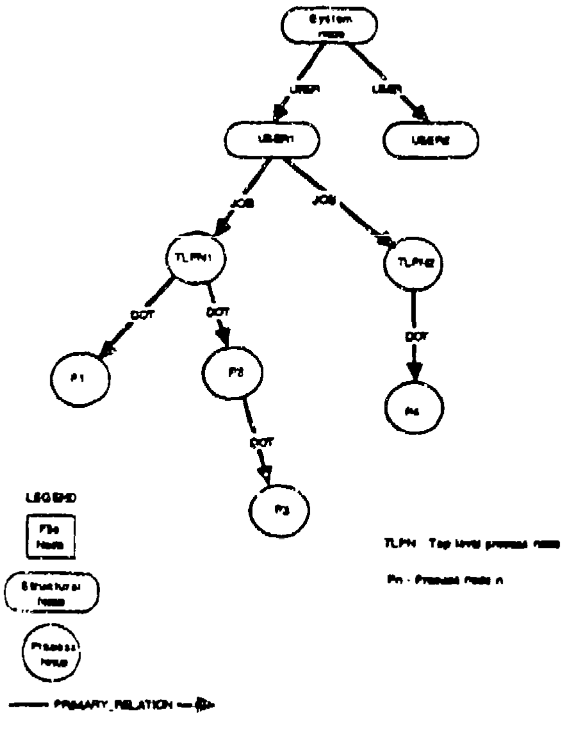


Figure 3(a). Process trees as seen by the CAIS.

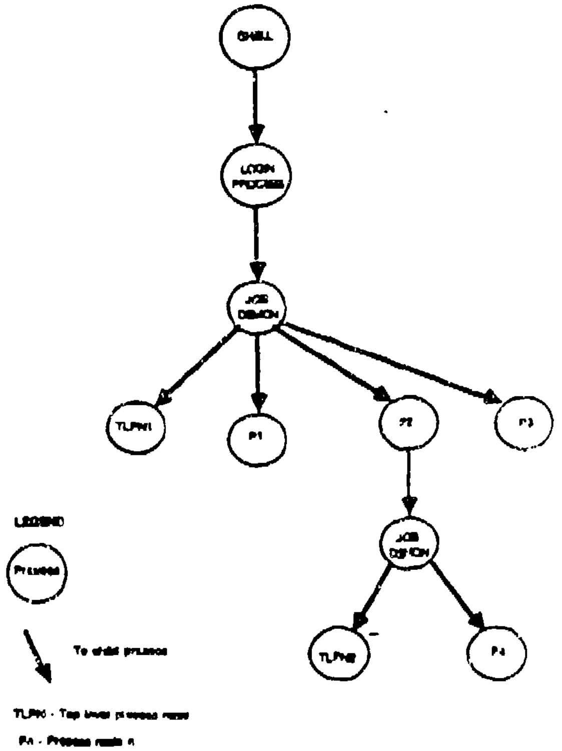


Figure 3(b). Process trees as seen by Unix.

requestor must wait (with a time limit) until the node is closed. The waiting process will then be released with an open node handle.

Tasks in a process can contend for a node with other processes, with other tasks in the same process or, incorrectly, with themselves. In the absence of a common synchronization method for all these cases, a package based on events with unique identifiers was specified. Events can be signalled or waited on with a time limit. Waiting processes place the event identifier into a wait queue in the node private status, report to the job server and wait on the event. Whenever a process closes the node, it scans the wait list for compatible requests, signals all such events and removes the entries.

The event services were to be re-implemented in the most efficient way available on the host system.

4.7 Access control.

Although it would have been possible to implement a CAIS environment without access control, it was considered essential to include it in our prototype in order to gain a good understanding of its features. The CAIS specification follows the DoD Document on Evaluation criteria for Trusted Computer Systems [4], referred to as the TCSEC. Both Discretionary and Mandatory access control are specified. Since neither of our targetted hosts implemented Mandatory access control, it was felt that implementation of such a feature without the underlying support from a protected domain was probably futile and misleading. Neither of the Host OS's in their present form can aspire beyond the C2 level, which is the highest level in the TCSEC categories with no mandatory access control.

Discretionary access control was implemented by imposing it as an extra restriction on existing host controls. This means that nodes would, in a final system, be inaccessible to all but the System Administrator or to processes logged in through the CAIS login facility. In our current working system, nodes are accessed using group rights.

Our implementation followed these guidelines:-

a) It was done literally, as described in the specification, using secondary relationships and their GRANT attributes.

b) All top-level user nodes are roles, with access relations to themselves, granting full access to adopting processes.

c) The top-level process, generated when a user logs in to the environment, is automatically given an adopted_role relation to its user node.

d) Top-level nodes and their access relations are created offline. When a process node is created, it is given a relationship to each top-level node or role node to which access was specified. The relations are USER to user nodes, DEVICE to all device nodes and ALLOW_ACCESS to all other accessible nodes. The ACCESS relation runs from the object to the user node. This, by itself, does not provide a path to the object.

e) When a process node is created, it is provided with an adopted_role relationship to its executable file node.

Figure 1 shows a node tree with one user and a role node. This role node holds a tool which has been invoked by the top-level process node. Only access control and primary relations are shown. The access relation from the user node to itself and the adopted_role relation from the cli to its own executable file node are not shown. Some experiences with access control are described later.

4.8 Process_control.

The implementation of process control marked the greatest divergence from the goal of portability. The MPX-32 system does not support process trees, whereas the UTX/32 system supports UNIX process trees which are similar to the CAIS trees. Therefore, UNIX facilities were used to ease the implementation of process control on that system.

The package was implemented using two demons. These are UNIX processes communicating through sockets with the actual CAIS processes. The main process control demon is the job demon. This controls all processes generated in a single job tree. There is also a resume-suspend-abort demon (the RSA demon).

4.8.1 The JOB demon.

The user logs into the environment by starting a login process from the UNIX shell. The login process sets off a job demon, having established a root process node and its relations under the user node. The job demon is passed parameters (using the host OS mechanism) which indicate the node pointer for the top-level process node and the file node for its executable image. The job demon then initiates the top-level process and waits for communication via its socket.

Processes communicate with the job demon when:-

a) a node is opened, closed or the open intents are changed.

b) another process is to be invoked or spawned.

c) an offspring exits.

The job demon detects exits via the signal generated by UNIX when a child process exits or aborts. It makes sure that nodes are not left open or with pending open requests in their private status.

The create_job service makes the new process tree into a direct offspring of the current process, not of the current job demon. This prevents the new job demon from being entered in the old job demon's list of process nodes. In this way, if the user logs out, the old job demon removes all offspring processes but does not remove the job demon created by the create_job service. Note that this job demon is left parentless in the host system - this is possible in UNIX but not in the CAIS. The new job demon controls its tree in the same way as the other job demons.

It was decided that the CAIS was deficient in not specifying a DELETE_JOF service to remove completed batch jobs. The KIT opinion, at that time, was that such job trees should be cleaned up outside the CAIS in the same way as the original login job tree. A DELETE_JOB service was added pending settlement of this issue.

When a job demon has no offspring, it closes all open nodes and then exits. If the login process is waiting on the job demon, it awakens and prompts for a user name.

The contrasting UNIX and CAIS process trees are shown in fig. 3.

4.8.2 The RSA demon.

The CAIS allows processes to resume, suspend or abort any process to whose node the current process node has the necessary access rights. Since the UNIX host does not allow unprivileged processes to do these things to processes with a different user identifier, this is done through an intermediary, the RSA demon. There is only a need for one such demon per host system even if a number of separate node models are being used simultaneously. The CAIS process control code verifies the CAIS rights and converts from a node pointer to a host process identifier (held in the victim process node private status). This identifier is sent, with the change of status request, to the demon which performs the action.

4.9 I/O and terminals.

The similarity between the CAIS I/O and the Ada I/O packages leads one to believe that an implementation of CAIS I/O on top of the Ada I/O packages might be feasible. However, certain differences in approach between the packages indicate that difficulties would arise if this course were taken. In particular, I/O to terminals has facilities such as prompts, intercepted characters and function keys, which made it necessary to deal with terminals independently at many points within the code.

The CAIS I/O packages were implemented down to a low-level set of subprograms below which the CAIS I/O would be host-dependent. Fortunately, most of the I/O facilities available on the UTX/32 system have equivalents on the MPX-32 host.

Problems with the use of file handles and node handles in the proposed MIL-STD-CAIS have been well publicized. Re-implementation with the scheme recently proposed [9] should make our implementation more elegant, since our file handles keep pointers outstanding to the node handle used to open the file even when the node handle is closed. This ensures that open path information is available at all times. In the new scheme, it will not be possible to close the node without also closing the file, so the CAIS access synchronization cannot be subverted.

Terminal I/O needs a number of unique features from the host OS. The Get subprograms must be capable of proceeding with execution if no characters are available in input buffers. The OS supported features for line editing must be by-passed so that intercepted characters and function keys can be specified as described. This means that raw I/O (binary I/O) must be used in the UTX/32 system.

The implementation dealt with text_io for terminals as though they were files using the host OS's line-editing and device independence facilities. This was a mistake, since there is no protection against a user's mixing text_io and scroll/page_terminal services on the same terminal. Since text_io used processed I/O and the terminal packages used raw I/O, the packages did not mix well. For a first implementation, this was tolerable so long as the user took care to use only support functions of text_io with the terminal packages. Support functions such as set_input, set_output, open and close are generally compatible in that they do little or no I/O to the terminal. At a later date, it is intended to make text_io for terminals use raw I/O and perform its own line editing.

In order to map available terminals onto the CAIS virtual terminal models, the UNIX termcap (terminal capabilities file) feature was adapted for a specialized use. The CAIS list format (List_utilities package) was found to be ideal for specifying such a file in order to simplify parsing of the user's terminal description. Termcap commands were reduced to those necessary to support the CAIS facilities, and several were added to implement features directly.

The six standard file nodes required by any CAIS process were opened during Text_io package elaboration. This was found to be a considerable overhead for process start-up. Deferment of the opening of the nodes until first use was felt to be unwise since a process could then find itself locked out of its current_output a long time after successfully opening and using current_input. It was found that the elaboration overhead could be reduced considerably by working below the Node_management interfaces from Text_io. This is mainly due to the fact that all six standard relations usually point to the same node and that all six opens can be done at the same time as the first open on Standard_input.

4.10 Maintenance Facilities.

A utility generates the System node together with the static structure of top-level nodes which cannot be manipulated from within the environment. This utility is capable of generating multiple node models in the same host system with the restriction that no more than one node model can occupy a single host directory. Low-level node_management routines are used to build and modify nodes without regard to access synchronization.

A node mending utility was produced, which is capable of accepting a node primary name and removing outstanding open intentions from the private status. Repair facilities for the relation and attribute files have not yet been found necessary.

The prototype was integrated and tested without the benefit of a source level debugger, so switchable trace code was built in to the Ada source. All exceptions can be sent to standard output through a switchable reporting routine that outputs the exception name, the current subprogram name and a brief reason. The debug trace and exception reporting is switchable using flags in the UNIX environment which are program-testable.

Other tools found necessary for maintenance were dump tools for relation and attribute parts of the node.

5.0 A CAIS Test Suite.

The strategy of prototyping implies that at least two versions of the software are planned. In reality, it was foreseen that even the prototype would go through several versions as performance was enhanced and errors were corrected. A test suite was seen as the best way to verify that successive implementations were still correct. A test manager was specified and designed, which would run as a CAIS process. This test manager would interpret scripts or accept commands interactively, run tests, log results and format them.

A set of approximately 700 tests were originally specified. Of these, time and manpower allowed the implementation of 95. The large size of CAIS processes in our implementation quickly caused a disk space problem. This was solved, temporarily, by linking the tests by package into test drivers. These drivers accepted a CAIS process parameter from the parent test manager and branched to the test specified by that parameter.

All tests had to be self-contained and built their own structures in a working node directory as required. The test manager cleaned them up after the test completed.

6.0 Command Language Interpreter and Tools.

The first Command Language Interpreter produced during development was a simple, interactive, exerciser for the Node_management and Process_control packages.

A more formal Command Language Interpreter was later produced, which accepts commands based on Ada language features. For example, a tool invocation for a copy tool resembles an Ada subroutine call:-

```
copy (from=>pathname1, to=>pathname2)
or
copy (pathname1, pathname2)
```

where both alternative sets of parameters are in the CAIS List format and can be passed more or less intact to the tool. The CLI allows minor relaxations from the strict List format. For example, quotes (") around pathnames and outer parentheses around the parameters may be omitted.

Most CLI features have tended to parallel in Ada those supplied by the UNIX C-shell, which supports a Job Control Language similar to the "C" language. So far, the CLI has been implemented entirely with CAIS subprogram calls and so is, theoretically, a portable tool.

A toolset was developed, mainly of small database management tools such as Create, Copy, Rename, Import, etc. for the display and maintenance of the nodes. A consistent toolset parameter interface was specified and a small Ada package of tool-building subprograms was built.

A group of UNIX tools were brought into the CAIS environment as "alien" tools. This was done in anticipation of a demand for sophisticated tools during the early lifetime of the CAIS. The vi editor, the UNIX print program and the ICC Ada compiler and linker were imported. Of these, only the print program was fully functional. A CAIS tool was written in Ada to interpret the parameters from the CLI, convert CAIS pathnames to host pathnames (by subverting CAIS interfaces) and then activate the alien tool as an offspring process in the host's process tree. The process node for the CAIS tool took the place of the actual tool in the CAIS process tree.

The vi tool was functional except that reads and writes from within the editor used host pathnames. In a real tool, these commands would have to be suppressed or modified. The Ada compiler and linker were imported for demonstration purposes only, since the whole library management file set was external to the CAIS and only the source and executable files were held in the CAIS environment.

7.0 Experiences with the CAIS.

7.1 Access control.

When our first working multi-user environments were created, the large number of access relations became an immediate problem. The desire to share access to tools led to the introduction of the group roles provided by the CAIS specification.

Access to tools in a user's own user tree is controlled when the user creates the tool. Access control lists can specify the access rights to be granted to the current user, other users and to group roles.

Sharing tools by granting access to other users or by making the tool executable file nodes into group roles rapidly becomes a painful overhead in any system as the number of users increases. Sharable tools are better placed as offspring of group roles as shown in Figure 1. Read access to the group allows the user to traverse to the group. Read and execute access from the tool to the group gives adopters of the group the right to execute the tool. To add tools to the group, a user needs append_relationships access to the group.

Access to other user trees as an automatic right is awkward, since membership of a common group gives access to nodes off the higher members of the group tree. In order to work down such a tree it is necessary to use the secondary potential_member relation to invert the tree. Each user wishing to access another user's nodes must become a potential_member of that user, so mutual sharing generates a web of potential_member relationships.

If more than two users wish to share trees, it is more efficient to make the potential_member relationships from the common group roles into two-way relationships. This is shown in Figure 2. This means that for every relationship which makes a user into a potential member of the group, another relationship makes the group a potential member of the user. In this way, each user can access another user's nodes by a two stage adoption (adoption is not a high overhead for a process). First, the group is adopted using the allow_access relation. Next, the other user is adopted by traversing the downward potential_member relation, an action validated by the upward potential_member relation. Using reciprocal potential_member relationships, the number of such relationships in a group of n users is reduced from n^2 to $2n$. The situation in

Figure 2 allows two users to adopt each other's user role.

Access control is an area of the CAIS which is easily described using relationships. However, the costs would rapidly become prohibitive even with group access rights unless real environments used compressed representations of the model. Static access control structures can easily be cached within each process to increase performance.

7.2 Tool building.

The KIT members have correctly predicted that CAIS implementations will tend to generate their own higher level CAIS packages for tool-building.

Examples of subprograms found useful to our implementation are:-

user_key- obtain the key of the current user (the username).

make_access_list - build a default access control list for node creation.

replace_wild_cards - replace wild cards in a pathname so that standard CAIS facilities such as path_key will accept the path.

remove_relation_and_key - split off the last relation and key from a pathname ignoring wild cards.

Considerable support was needed for the Iterator services, which have often been criticised for their minimal functionality and apparent inefficiency. We have made proposals for extra iterator services, but it is likely that no iterator interface will find complete acceptance in the user community.

7.3 Tool portability.

A set of tools were supplied by the Mitre corporation from their prototype CAIS implementation [7]. These were ported onto our environment as an experiment in portability. Their group of node management tools ported directly on to our environment with no changes except for "with" and "use" Ada package statements.

The other two tools supplied were an interactive menu manager called Video and a line oriented text editor called Aled. Both programs were originally obtained from the Ada repository. The Aled tool used Scroll_terminal interfaces and the Video tool used Page_terminal. Unfortunately, at that time, the Mitre implementation of these packages was only partial and it was necessary to modify both tools to handle the different approach to input of control characters in the full implementation. The tools ported easily once the input sections were recoded to use function keys.

8.0 Performance.

The first demonstrable version of the prototype was available in December 1985 and was, as expected, very slow. Log in to the environment took an average of 20 seconds and tool invocation took an average of 25 seconds.

A short performance enhancement project of 3 months was undertaken, in order to have a publicly acceptable demonstration by April 1986. The areas of enhancement were:-

a) Templates were used foı process node relations and attributes files so that simple file copies could be used to generate the process nodes. The templates were built by the Generation utility used to build the static top-level node framework.

b) Each process node has a group of pre-defined relations which can only be modified by calls to CAIS interfaces. The node pointers for these relations were cached so that the open services no longer needed to search the relations file to access the nodes.

c) The open and spawn services are the points at which any inefficiency in the implementation is perceived. Much of spawn's time is spent in the open service, therefore close attention to this service is the key to increased efficiency.

1) traversal of node relationships can be improved by use of low-level interfaces rather than formally opening each intermediate node for read_relationships and closing the same node later. If lock-out occurs, the traversal code backs up to call the full CAIS open service so that access synchronization to the intermediate node works correctly.

2) access control potentially involves searches of trees of roles. Considerable care must be taken in the implementation of these searches. The role trees are static since PERMANENT_MEMBER relationships cannot be manipulated by users. Therefore cached or reformatted access control information should be used.

d) Analysis of the quantity of I/O performed on node parts showed immediate and spectacular areas for improvement.

The version demonstrated, in April 1986, to members of the KIT/KITIA and to other interested parties showed a fourfold increase in performance for the two measures, entry into the environment and tool invocation. The times are compared in table 1 below. Spawning time for a tool is the time between a call to spawn and the completion of the new process elaboration. The exit time is the time from the last Ada statement to the end of await_process_completion in the parent process.

Version 1.0 is the December 1985 version and version 1.2 is the April 1986 version. All times were measured on a stand-alone Gould 9080 system. The software was built using the ICC 3.1 Ada translator running on UTX/32 (the UNIX host).

Table 1. Sample performance measurements for the Gould CAIS prototype.

Measurement	Version 1.0	Version 1.2
Entry into the environment.	18.0s	4.2s
Spawning a tool.	25.5s	6.9s
Exiting from a tool.	11.9s	1.3s

9.0 Size.

Table 2 shows a list of CAIS package sizes expressed as number of lines of code (whole line comments excluded), number of statements and number of bytes of storage for code generated by the ICC Ada translator on a Gould system. The number of statements was based on a count of semicolons not found in strings or comments. It is intended to show the relative complexity of coding of the packages.

Notice that the process control line count includes the code for the job and RSA demons, while the byte count does not, since these are in separate executable files of no great size.

The CAIS, implemented as a non-shared library, is a considerable overhead for any real APSE environment. For every tool in the environment, more than 1 Megabyte of disk space must be reserved. Also, small memory configurations suffer considerable degradation because of increased swapping as more users enter the environment. Our development machine was configured with 16 Megabytes of memory, but most users at this time do not have such luxury.

Table 2 Line counts and storage for CAIS packages.

Package	Lines	Statements	% Stat-ements	Bytes	% Storage
Access_control	1,866	1,222	4.5	25,296	2.1
Attributes	3,201	1,578	5.8	34,162	2.9
File I/O	7,983	4,764	17.5	154,172	12.9
Direct_io	-	-	-	29,192	-
Io_control	-	.	.	12,884	.
Sequential_io	-	.	-	31,262	-
Text_io	-	-	-	80,878	-
Import_export	470	302	1.1	10,496	0.9
List_utilities	3,343	2,082	7.7	124,640	10.5
Node_management	16,146	8,720	32.1	398,768	33.5
Structural_nodes*	.	.	.	.	.
Page_terminal	1,756	1,046	3.9	97,736	8.2
Process_control**	8,852	4,573	16.8	64,808	5.4
Scroll_terminal	1,265	767	2.8	59,024	5.0
Parsing routines Debug Common routines	2,896	1,449	5.3	16,648 2,872 19,576	1.4 0.2 1.7
CAIS Ada code	47,881	26,491	97.5	1,008,388	84.7
CAIS Ada data				174,116	14.6
"C" routines	1,737	672	2.5	8,680	0.7
CAIS total	49,618	27,163	100.0	1,191,184	100.0

* included in node_management total

** line count includes demons, byte count does not.

The use of a node server to perform all access to the node model would reduce the amount of code held in each tool as would the introduction of shared Ada libraries. Ideally, such shared libraries would be placed in some kind of protected domain so as to improve the credibility of the CAIS as a Trusted Computing Base. It has been pointed out [8] that such shared libraries would introduce minor technical problems in the area of generic packages in the CAIS.

10.0 Future Plans.

The prototype source has been offered for public distribution at a nominal price for evaluation, tool-building and for education. Other software such as our Test suite, toolsets and CLI are regarded as proprietary at this time.

The immediate plans for this project are to implement the remaining packages to attain as near to 100% functionality as is practicable. The remaining packages are Magnetic_tape and Form_terminal.

It is intended that the environment be ported onto a Secure Unix product produced by this company. This version of UNIX is currently being evaluated for compliance with the C2 level as specified in the TCSEC [4].

11.0 Conclusions.

We feel that the following conclusions can be drawn from the project:-

1) the CAIS is implementable on a UNIX host. We are reasonably confident that our design could be ported to our own Real-Time Operating System.

2) the performance overheads are not prohibitive. The times quoted are stand-alone on a powerful machine, but it should be noted that this prototype was designed with only small regard for ultimate efficiency. Careful design, using prototyping experience, should enable us and others to produce an environment with an acceptable user response.

3) the CAIS does provide usable features. Much criticism of the specification has stated that interfaces are too primitive for tool-builders. We have found it possible to build useful higher-level services using the primitive interfaces in the specification.

4) the access control model, while complex, does appear to provide the features required to implement a full discretionary access control model to current DoD requirements.

5) the prototype is a useful tool for further investigation of the implications of the CAIS specification and it is stable enough to build working tools and toolsets in anticipation of future production environments. This should, in theory, reduce the lead-time before such environments support a worthwhile body of applications software.

It is hoped that some of the data in this paper are of some use to other potential implementers in their own estimates of time and man-power required.

Acknowledgements.

We would like to thank the CAIS specification team for all the help they have given us during implementation, in clarification and guidance. In particular we thank P. Oberndorf, T. Harrison and E. Ploedereder for their patience with our questions.

We would also like to thank the Mitre Corporation for supplying their initial toolset in order to evaluate an attempt to port the tools to a different CAIS environment.

References.

1) Military Standard Common APSE Interface Set (CAIS). PROPOSED MIL-STD-CAIS. Department of Defense.
January 1985.

2) CAIS Rationale. Institute for Defense Analyses, 1801, N. Beauregard St., Alexandria, Virginia, 22311.
April 1986.

3) The CAIS Reader's Guide. IDA Memorandum Report M-150. J.F. Kramer *et al.* Institute for Defense Analysis.
December 1985.

4) Department of Defense, Trusted Computer System Evaluation Criteria. CSC-STD-001-83. DoD Computer Security Center.
August 1983.

5) [ANSI 78] American National Standards Institute, Magnetic Tape Labels and File Structure for Information Interchange (ANSI Standard x3.27-1978).

6) Structured Design. E. Yourdan and L. Constantine. Prentice-Hall Inc. 1979.

7) A study of the Common APSE interface set (CAIS). Mitre Corporation Draft Working paper. R. Bowerman *et al.* Mitre Corp., 1820, Dolley Madison Blvd., McLean, Virginia 22102.
October 1985.

8) The CAIS package structure. E. Ploedereder. Communication to CAIS Working Group.
23rd. May 1986.

9) Proposed specification changes distributed at KIT/KITIA meeting. January 1986.

TRW CAIS Prototype

A Report to the KIT/KITIA

April 16, 1986

Structure of the Presentation

- History and Current Status
- Objectives of CAIS Prototyping
- Risks to CAIS Acceptance
- TRW's Prototype
  - Design Objectives
  - Selected Design Features
- Addressing the Risks

ASE

TRW's ASE Project Tasks

TRW

TRW calls its general KIT/KITIA support contract **ASE** (Ada Software Engineering)

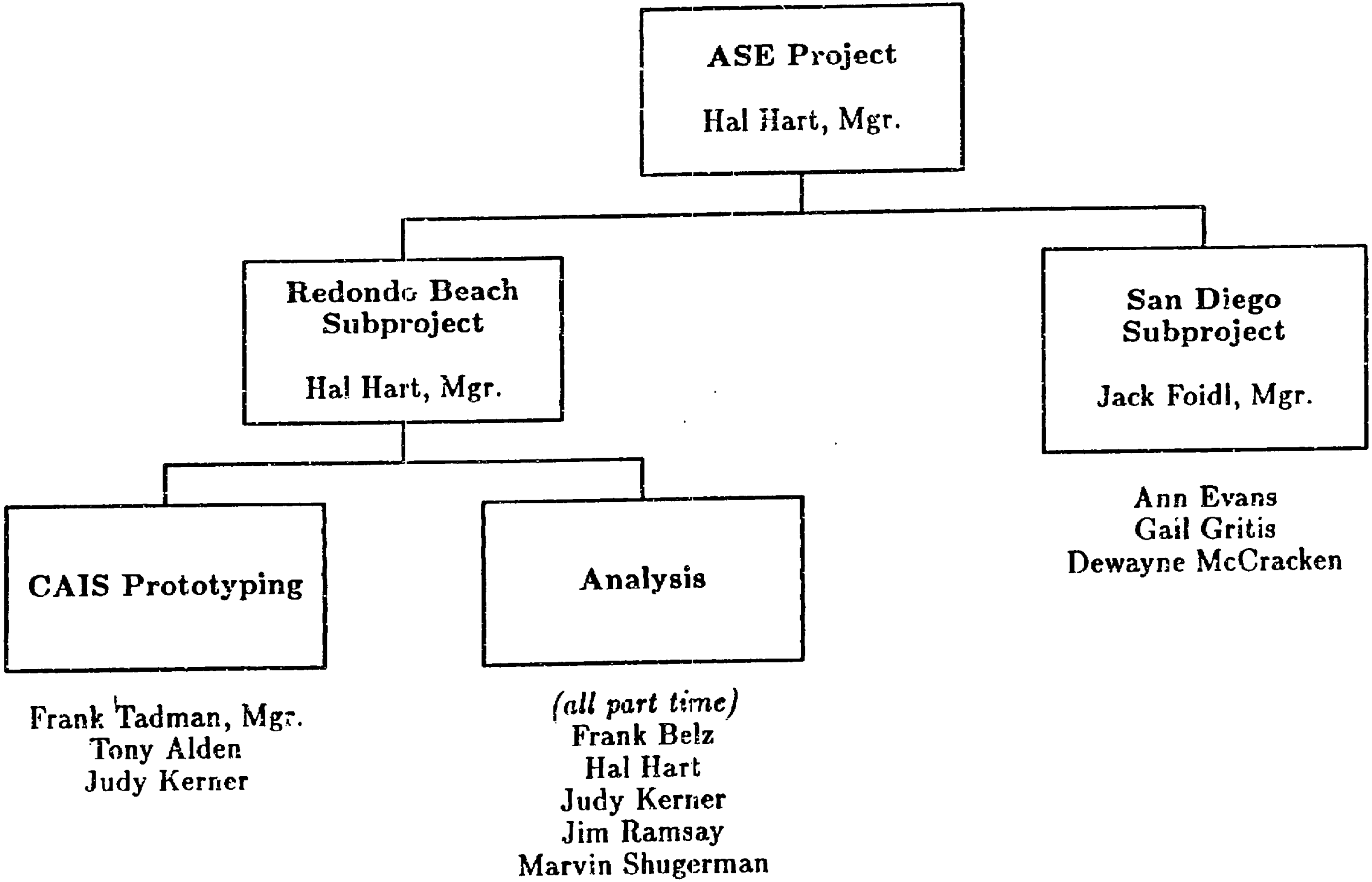
- CAIS prototyping and experimentation
- KIT management support
- CAIS Requirements and Criteria: analysis, rationale, and maintenance
- CAIS Specifications: maintenance
- Transportability Guide: complete it (new task)
- Software Systems Engineering
  - CAIS-to-RAC compliance
  - CAIS/RAC support for STARS-SEE (esp. security needs and completeness)
  - CAIS comments analysis
- KIT ARPANET/MILNET database and communications
- Support for other KIT/KITIA working groups

General KIT/KITIA support was also provided under previous contracts (including AdaPAKSS) since August 1981.

ASE

ASE Project Organization

TRW



ASE Project

Hal Hart, Mgr.

Redondo Beach Subproject

Hal Hart, Mgr.

CAIS Prototyping

Frank Tadman, Mgr.
Tony Alden
Judy Kerner

Analysis

(all part time)
Frank Belz
Hal Hart
Judy Kerner
Jim Ramsay
Marvin Shugerman

San Diego Subproject

Jack Foidl, Mgr.

Ann Evans
Gail Gritis
Dewayne McCracken

CAIS Prototyping Accomplishments

June 1983 – April 1985:

[Under previous contract AdaPAKSS (Advanced APSE Prototype and KAPSE Standardization Support)]

- Established evaluation criteria
  - CAIS (specification)
  - CAIS implementations
- Produced a high-level design of CAIS prototype
- Identified key design issues
- Wrote a low-level design for a basic CAIS subset using Ada PDL
- Submitted numerous CAIS comments
- Reported results to KIT/KITIA (January 16, 1985 — Frank Belz)

CAIS Prototyping Accomplishments (ASE)

February 1 — present:

ASE

- Reviewed prototype design, made minor modifications
- Implementing basic CAIS
  - Near completion:
    - ◇ **node_management**
    - ◇ **structural_nodes**
    - ◇ **list_utilities**
    - ◇ Supporting lower-level packages
  - In progress:
    - ◇ **attributes**
    - ◇ **text_io**
  - Limitations:
    - ◇ No access control enforcement
    - ◇ No exclusive access enforcement
    - ◇ No time limits
    - ◇ No iterators
    - ◇ All structures are built in main memory

Host Environment

- Sun 2 and Sun 3 workstations
- Sun Operating System, 3.0 release
  - Derived from Berkeley 4.2BSD Unix
- Verdix Ada Development System (VADS), version 5.1a

Objectives of CAIS Prototyping

- To verify implementability on commonly used hosts.
- To provide multiple bases for tool portability experiments.
- To explore implementation options in support of varying design goals.
- To improve understanding of the strengths and weaknesses of the CAIS.

Risks to CAIS Acceptance

- Performance
- Portability
- Security
- Appropriateness
  - CAIS interfaces
  - Underlying models

Performance

- The CAIS is more complex than typical current operating systems, therefore standard techniques for acheiving efficient operation may not be sufficient.
  - Efficient operation may require new algorithms and/or architectures.
- The efficiency of piggyback implementations is especially important.
  - In many current layered systems, a substantial performance penalty is paid.

Portability

- The CAIS must be available on many hosts, but reimplementing the CAIS on every host would be very expensive.
  - A CAIS implementation *itself* should be portable to a large class of machines with only a small part of it machine and operating system dependent.

Security

- The DoD is increasingly concerned with operating system support of military security policies.
  - The CAIS security mechanisms must be adequate to support these security policies.
  - The CAIS security mechanisms must be implementable with acceptable performance.
  - The security mechanisms must be usable by both the tool writer and the APSE user.

Appropriateness

- The CAIS interfaces may not support portability and functionality of tools.
  - The CAIS must not omit important interfaces nor provide inappropriate ones.
  - The CAIS must be usable by the tool writer.
- The underlying CAIS model may not admit clear and useful presentation to users.
  - The CAIS network is more difficult to visualize than a hierarchical filesystem.
  - The access control mechanism may be difficult to use. If a user is unsure of exactly what the permissions to an object mean, it is impossible to use discretionary access control effectively.

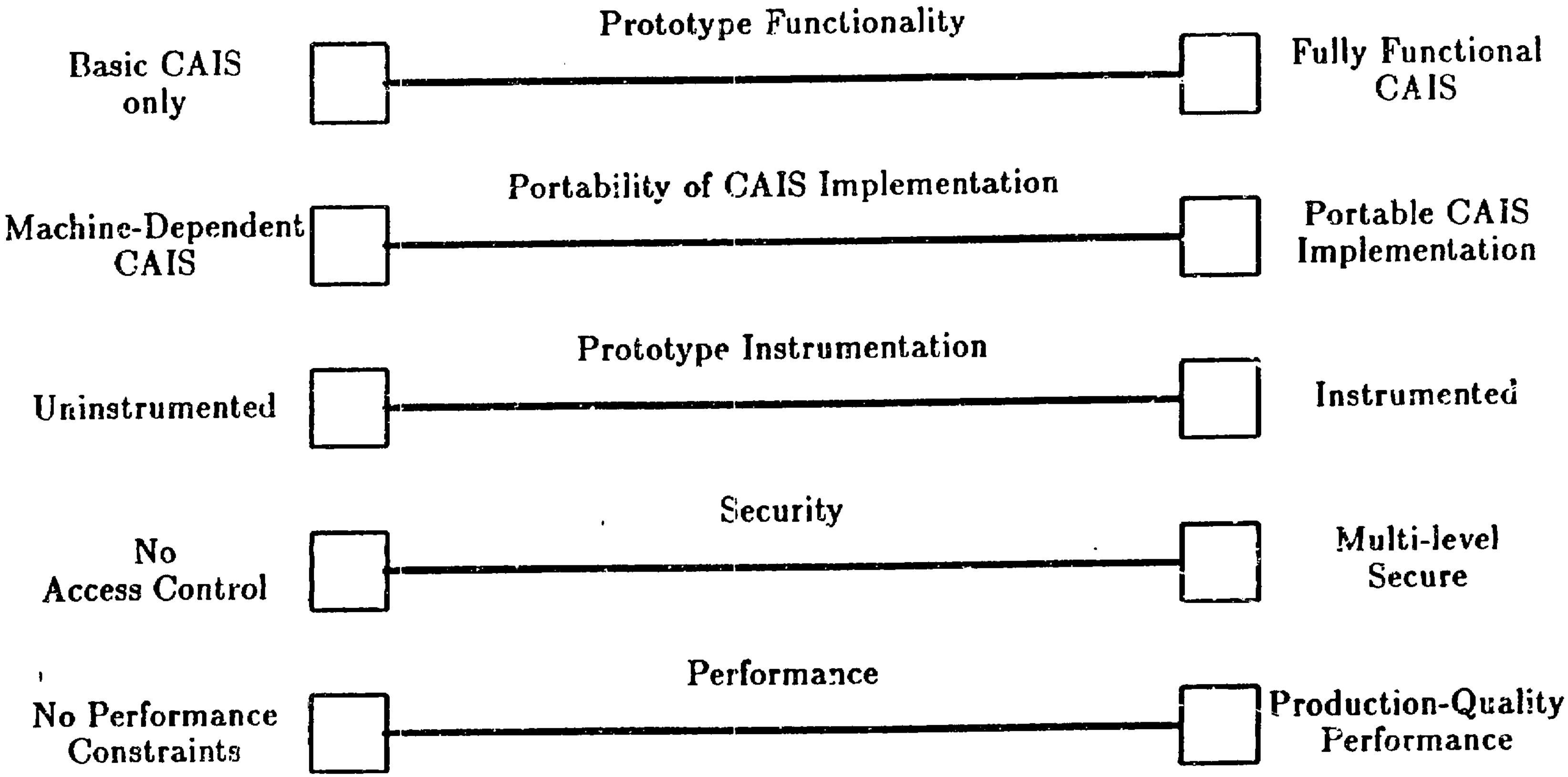
TRW's prototype

- A focused approach is needed that addresses critical risks which are not being adequately addressed elsewhere in order to maximize the CAIS program's overall payoff/cost ratio.
- A basic prototype has been designed and is now being implemented, which includes all of the CAIS functions absolutely necessary to support tools.
- Beyond the basic prototype, several possible dimensions of implementation have been identified and are presented on the following slides.

ASE

Ranges of Design Objectives

TRW



Prototype Functionality

Basic CAIS only — Fully Functional CAIS

Portability of CAIS Implementation

Machine-Dependent CAIS — Portable CAIS Implementation

Prototype Instrumentation

Uninstrumented — Instrumented

Security

No Access Control — Multi-level Secure

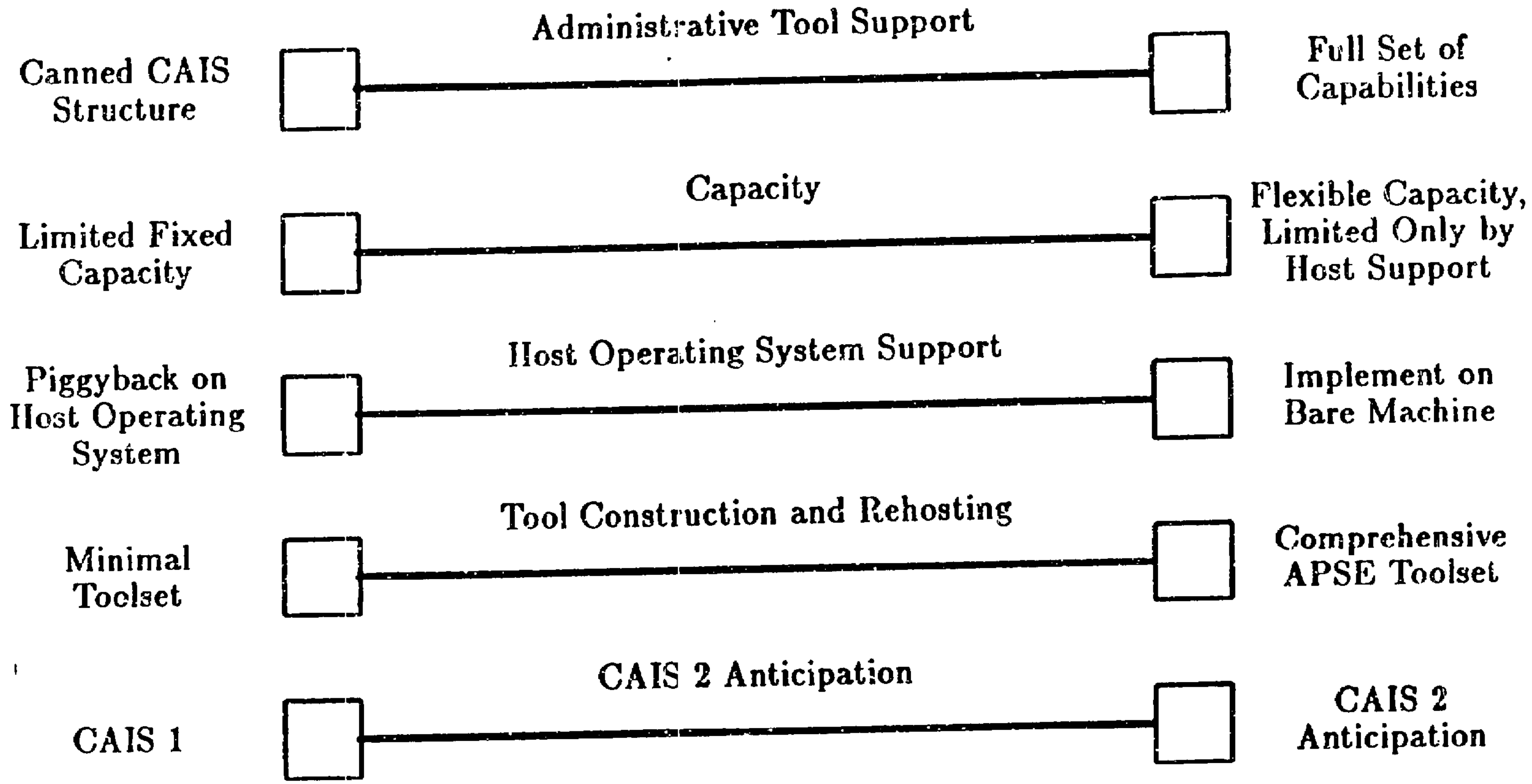
Performance

No Performance Constraints — Production-Quality Performance

ASE

Ranges of Design Objectives (cont.)

TRW

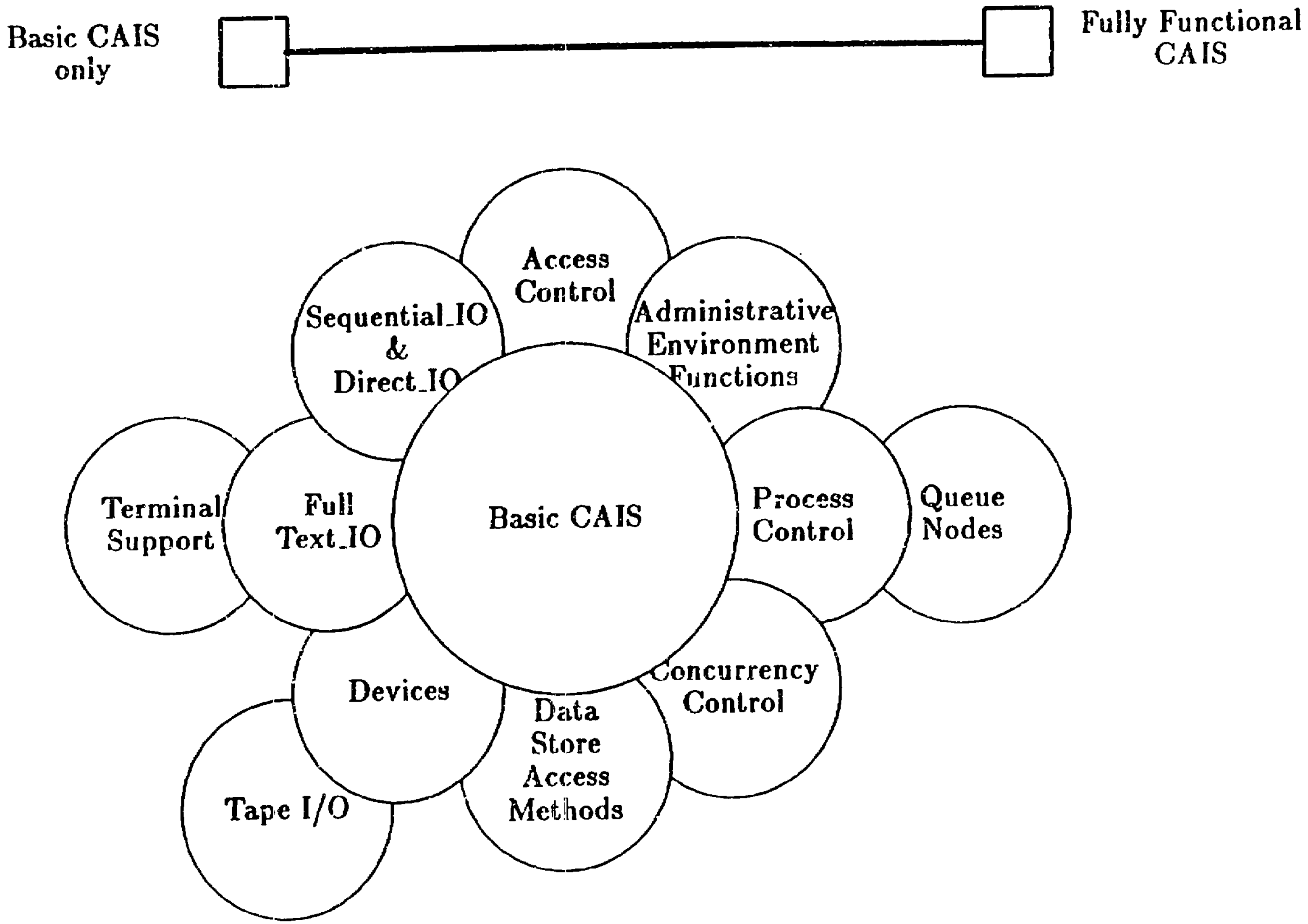

Administrative Tool Support
Canned CAIS Structure
Full Set of Capabilities
Capacity
Limited Fixed Capacity
Flexible Capacity, Limited Only by Host Support
Host Operating System Support
Piggyback on Host Operating System
Implement on Bare Machine
Tool Construction and Rehosting
Minimal Toolset
Comprehensive APSE Toolset
CAIS 2 Anticipation
CAIS 1
CAIS 2 Anticipation

Details of the Objectives

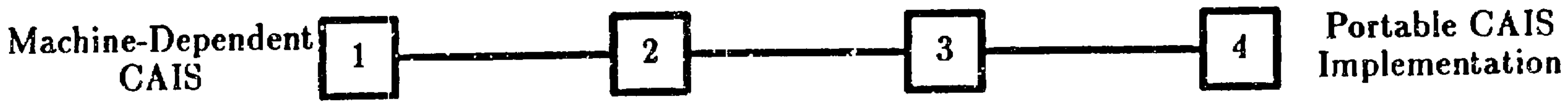
- The following slides describe the range of functionality for each objective.
- Each slide shows the endpoints of the scale and some sample points in between. These points are illustrations only; some objectives can be achieved in other ways.

ASE

Prototype Functionality



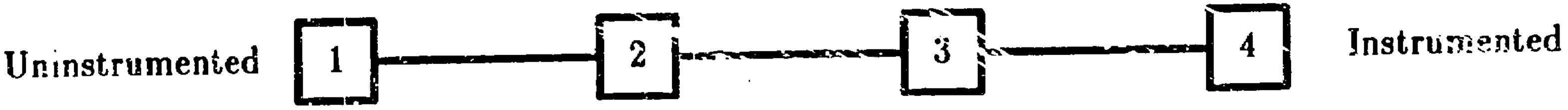
Portability of CAIS Implementation



1. No concern for machine independence
2. Portable to all Unix systems
3. Portable to standard mainframes
4. Small inner portability kernel

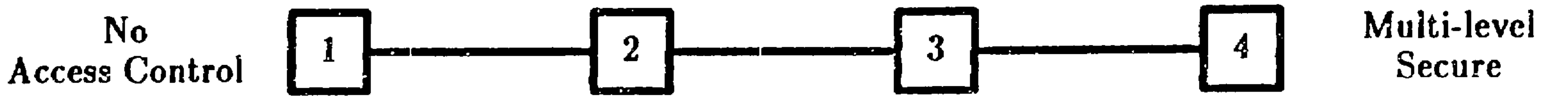
Prototype Instrumentation

3--634



1. No instrumentation
2. Automatic instrumenting tools
3. Selected CAIS-specific instrumentation
4. Full instrumentation

Security

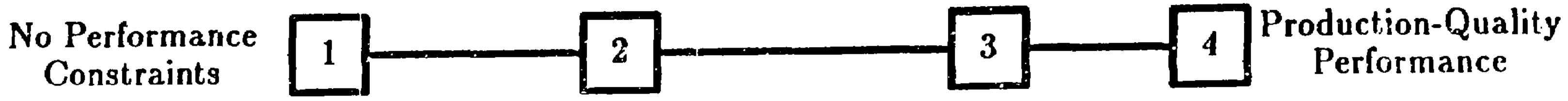


1. No access control
2. CAIS discretionary mechanisms and mandatory labels
3. Existing TCB discretionary and mandatory mechanisms
4. Multi-level secure with CAIS discretionary mechanisms

ASE

Performance

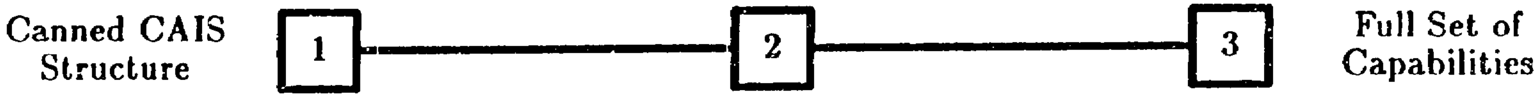
TRW



1. No performance constraints
2. Known algorithms/optimizations only
3. Optimal architectures/algorithms
4. Production-quality performance

ASE	Administrative Tool Support	TRW

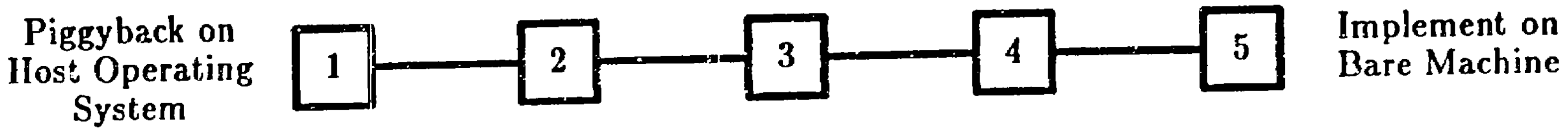
Note: This category of design objectives addresses those parts of the CAIS implementation for which CAIS interfaces are not now specified.



1. Canned CAIS structure (fixed set of users, groups, etc.)
2. Necessary support for CAIS 1 operation
3. Full set of interfaces and capabilities for all useful administrative tools (e.g. necessary support for CAIS operation, maintenance, archival, etc.)

ASE

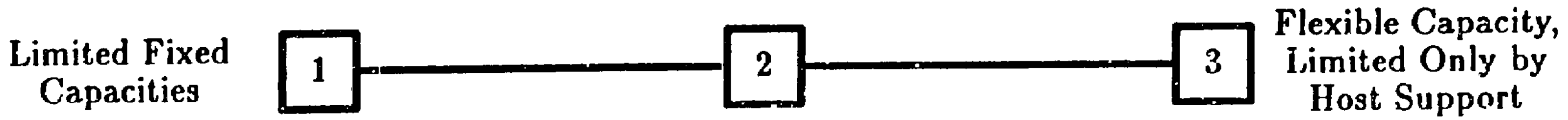
Host Operating System Support



1. Non-privileged host user using host filesystem
2. Privileged host user using host filesystem
3. Process using disk partition
4. In-kernel enhancements
5. Bare machine implementation

ASE

Capacity

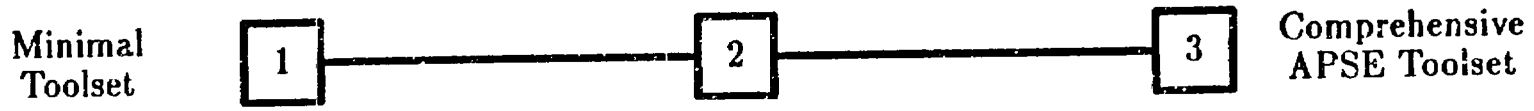


1. Limited, fixed capacity
2. Arbitrary, reasonable limits for production-quality CAIS
3. Flexible capacity, limited only by host support

ASE

Tool Construction and Rehosting

TRW



1. Minimal toolset
2. Demonstration of toolset philosophy
3. Complete APSE toolset

CAIS 2 Anticipation

CAIS 1
1
2
CAIS 2
Anticipation

1. Implement only CAIS 1 with no changes
2. Implement some CAIS 2 functionality as it is proposed

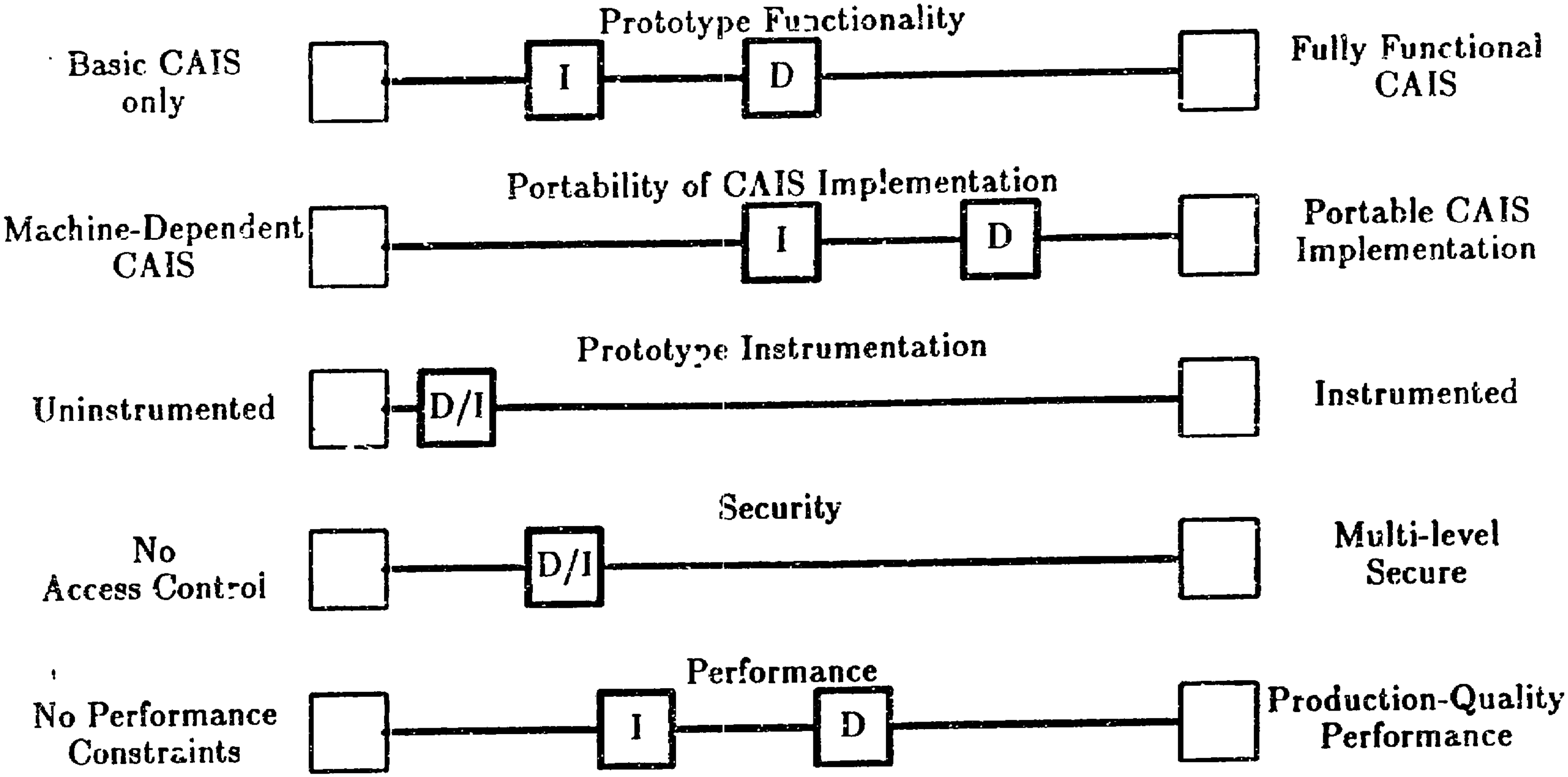
- Possible areas of investigation
  - Distribution
  - Strong typing
  - Access control
  - Transaction mechanism

ASE

Status of Prototype Design and Implementation

TRW

3-642



Prototype Functionality

Basic CAIS only

I

D

Fully Functional CAIS

Portability of CAIS Implementation

Machine-Dependent CAIS

I

D

Portable CAIS Implementation

Prototype Instrumentation

Uninstrumented

D/I

Instrumented

Security

No Access Control

D/I

Multi-level Secure

Performance

No Performance Constraints

I

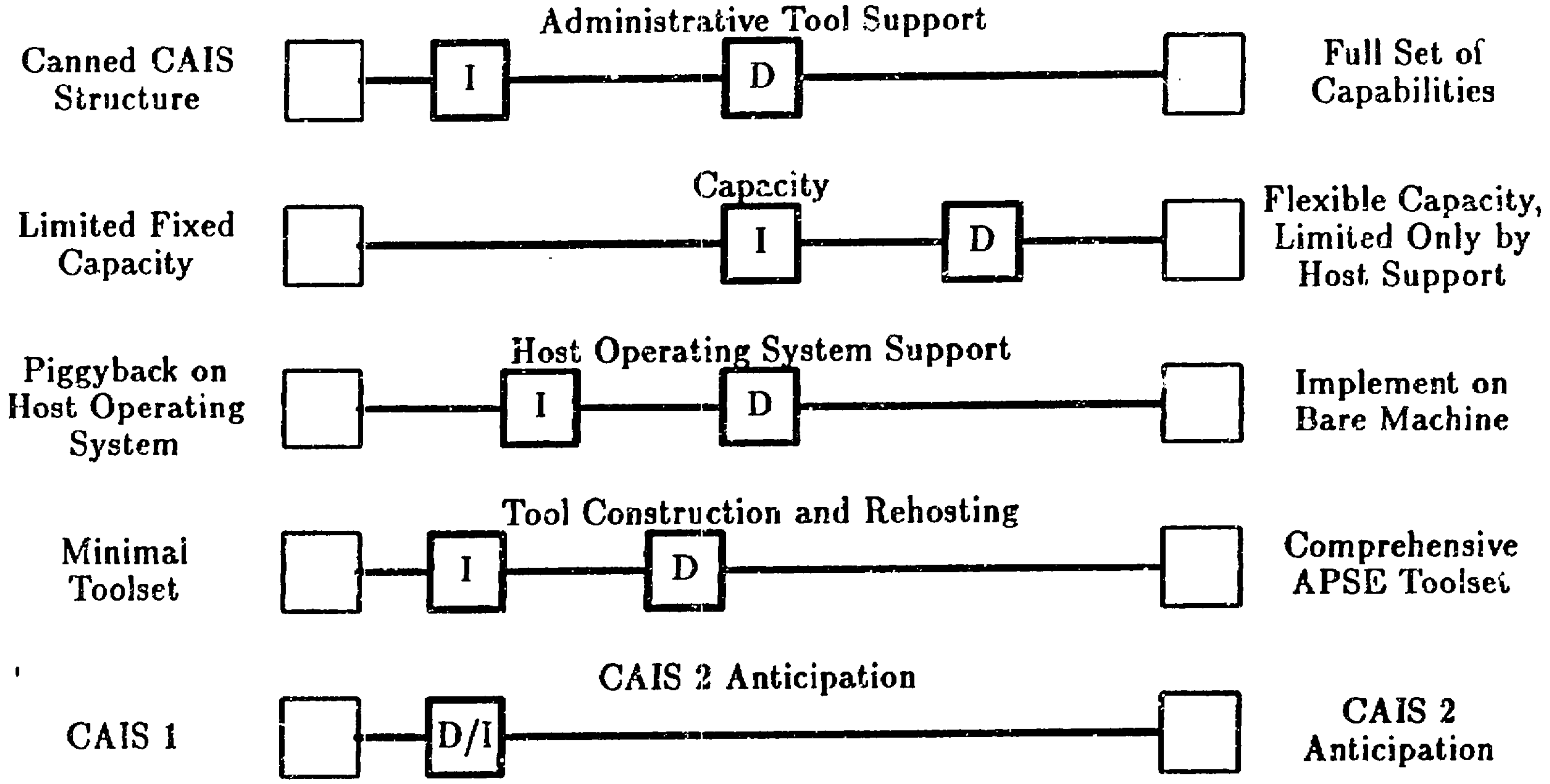
D

Production-Quality Performance

ASE

Status (cont.)

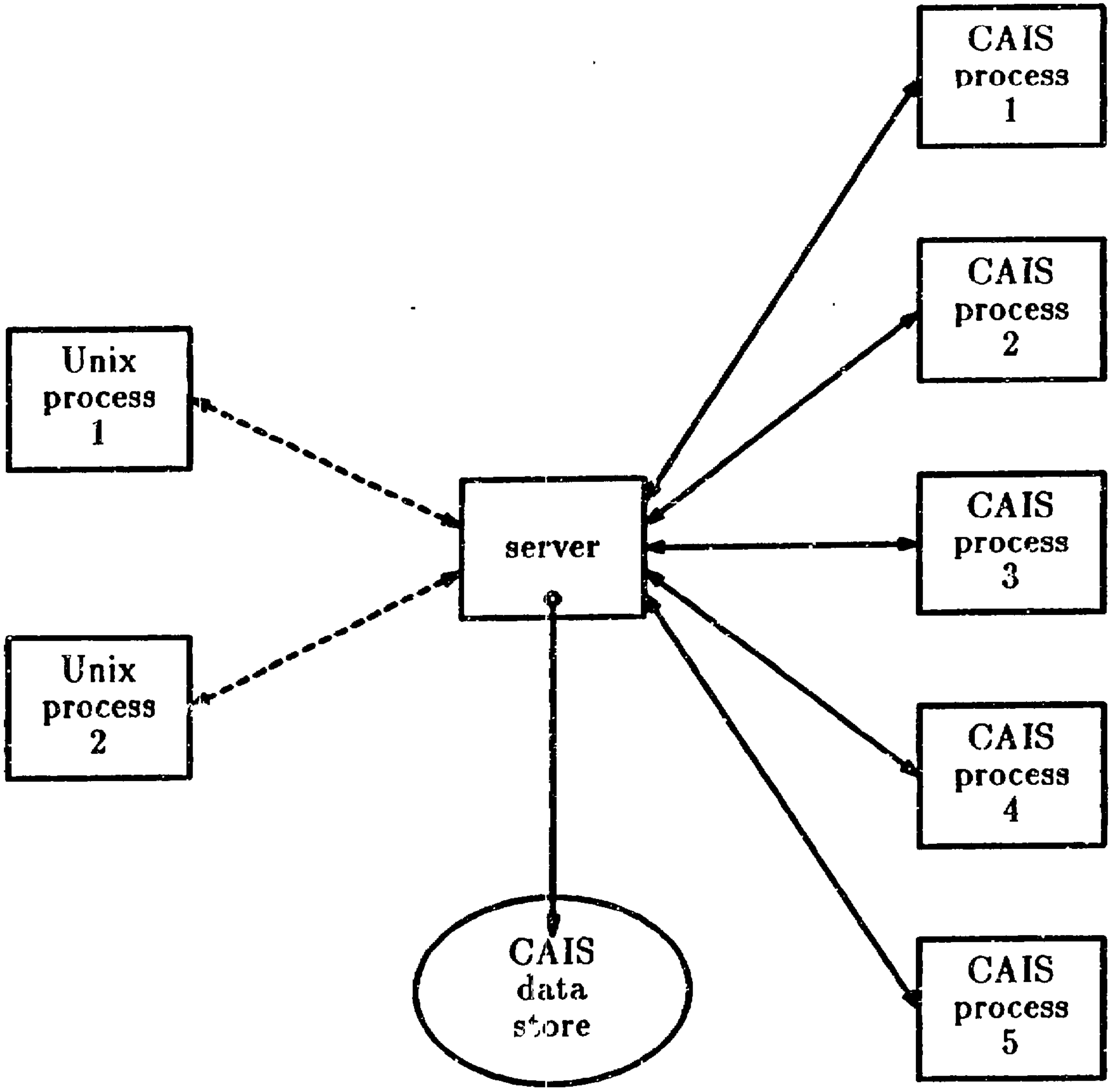
TRW


Administrative Tool Support
Canned CAIS Structure
I
D
Full Set of Capabilities
Capacity
Limited Fixed Capacity
I
D
Flexible Capacity, Limited Only by Host Support
Host Operating System Support
Piggyback on Host Operating System
I
D
Implement on Bare Machine
Tool Construction and Rehosting
Minimal Toolset
I
D
Comprehensive APSE Toolset
CAIS 2 Anticipation
CAIS 1
D/I
CAIS 2 Anticipation

ASE

Process Structure

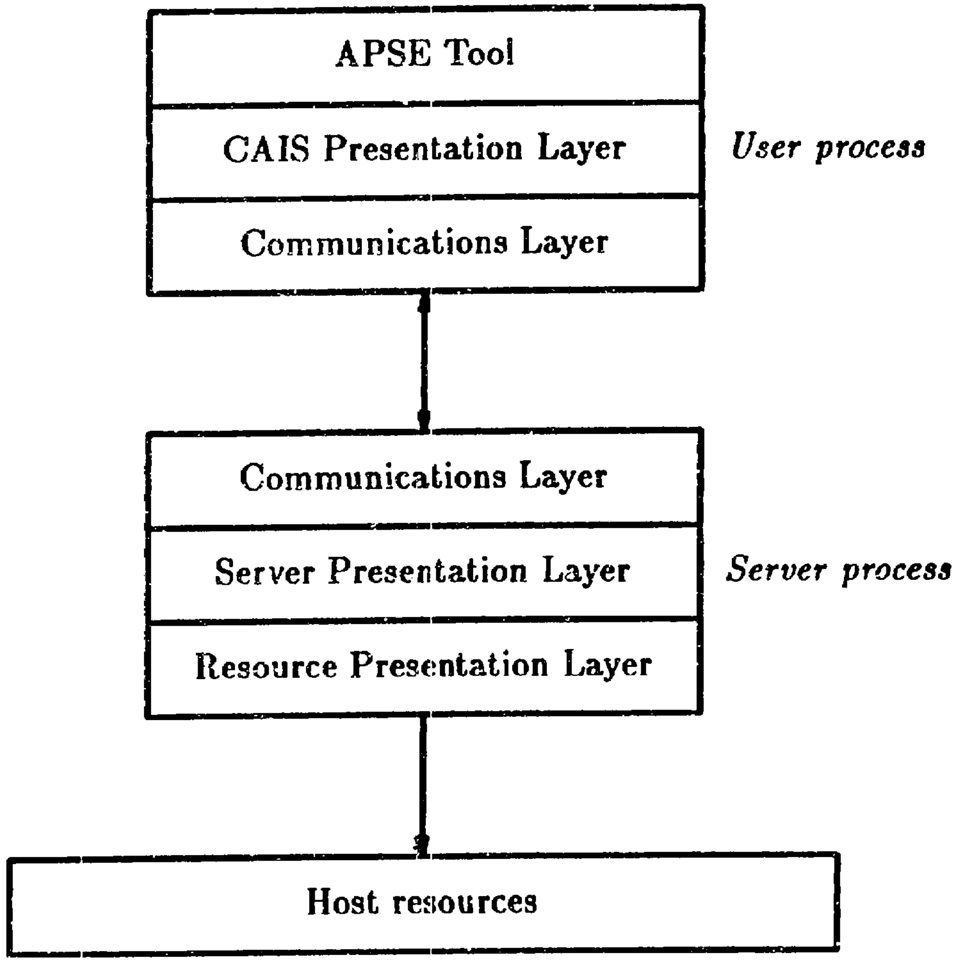
TRW



ASE

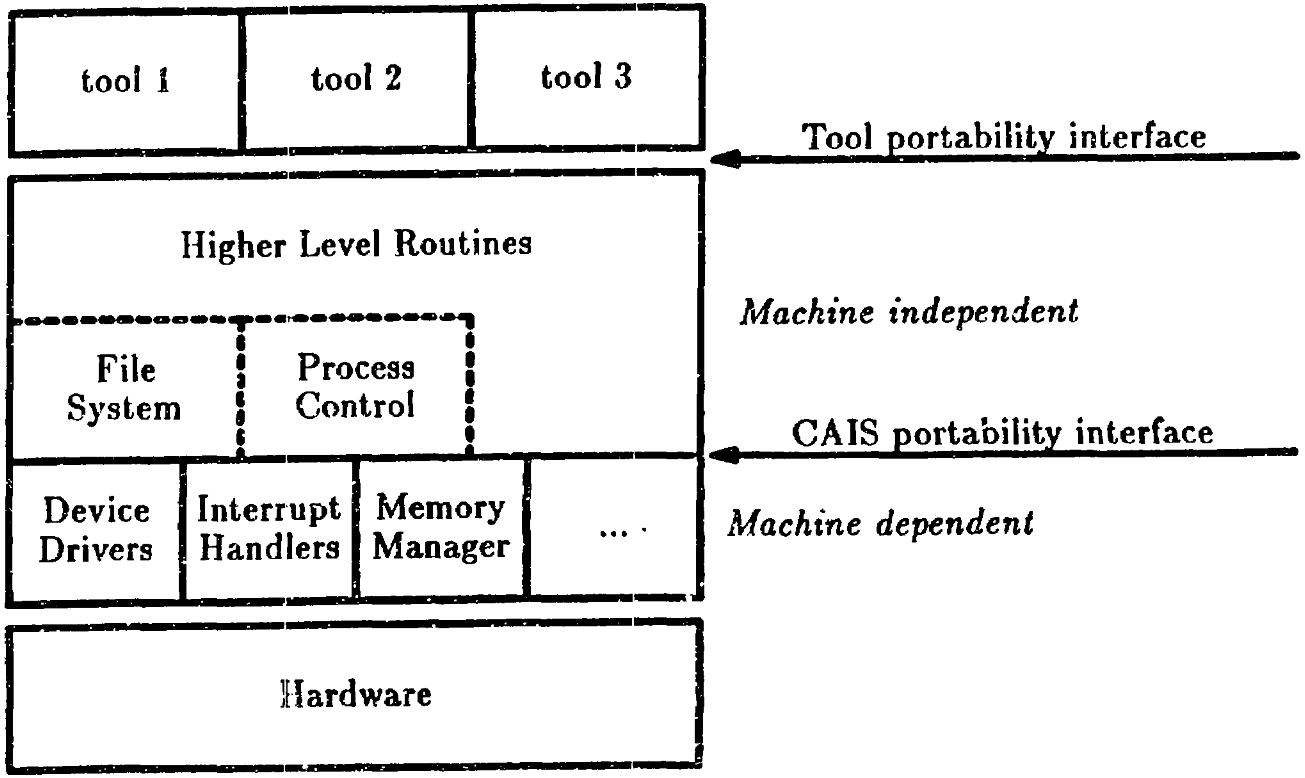
Layer Allocation to CAIS Processes

TRW



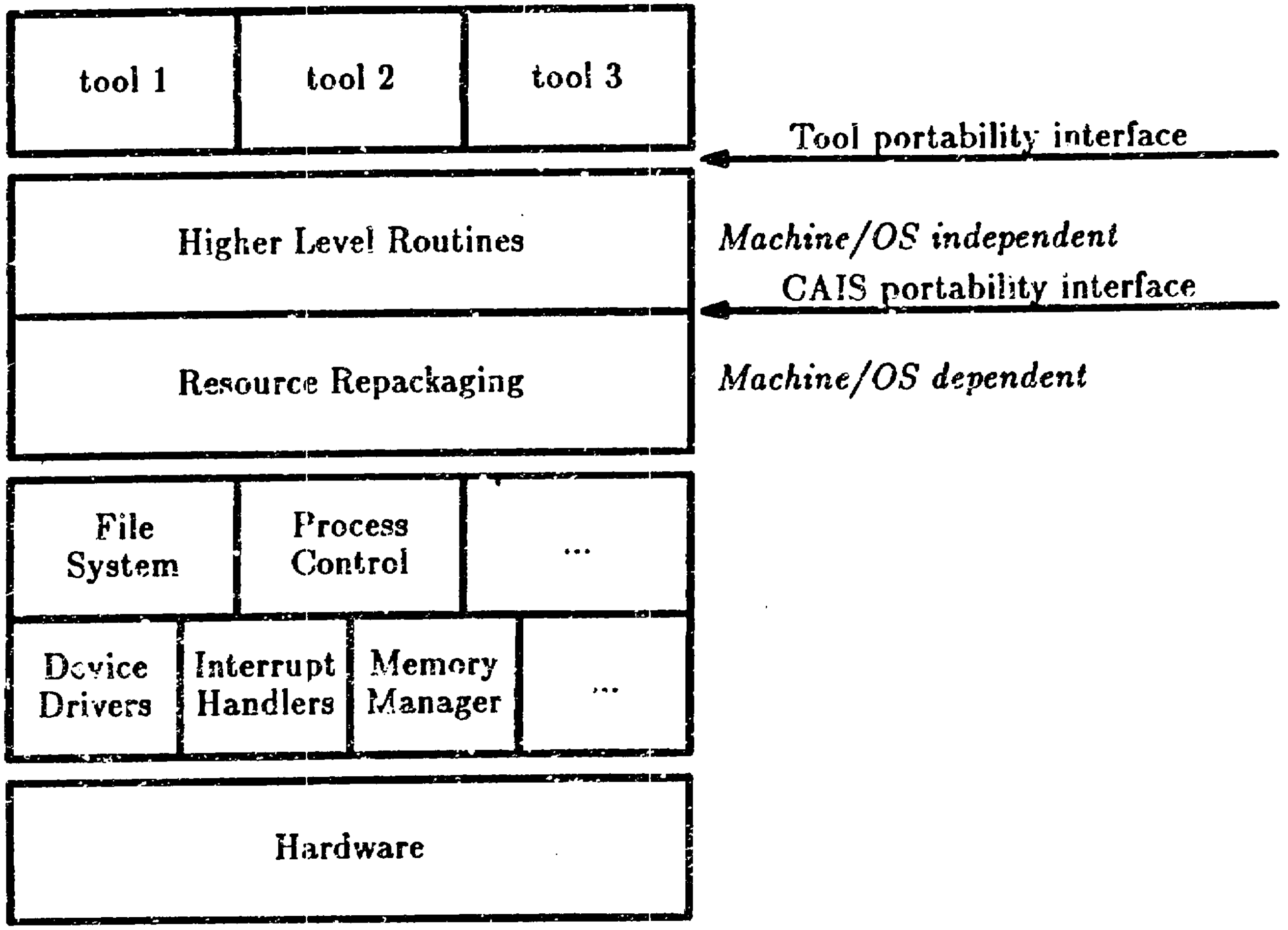
Inner Portability Layers (bare machine)

- Hardware resources are repackaged by lower hardware-dependent layers into hardware-independent resources.



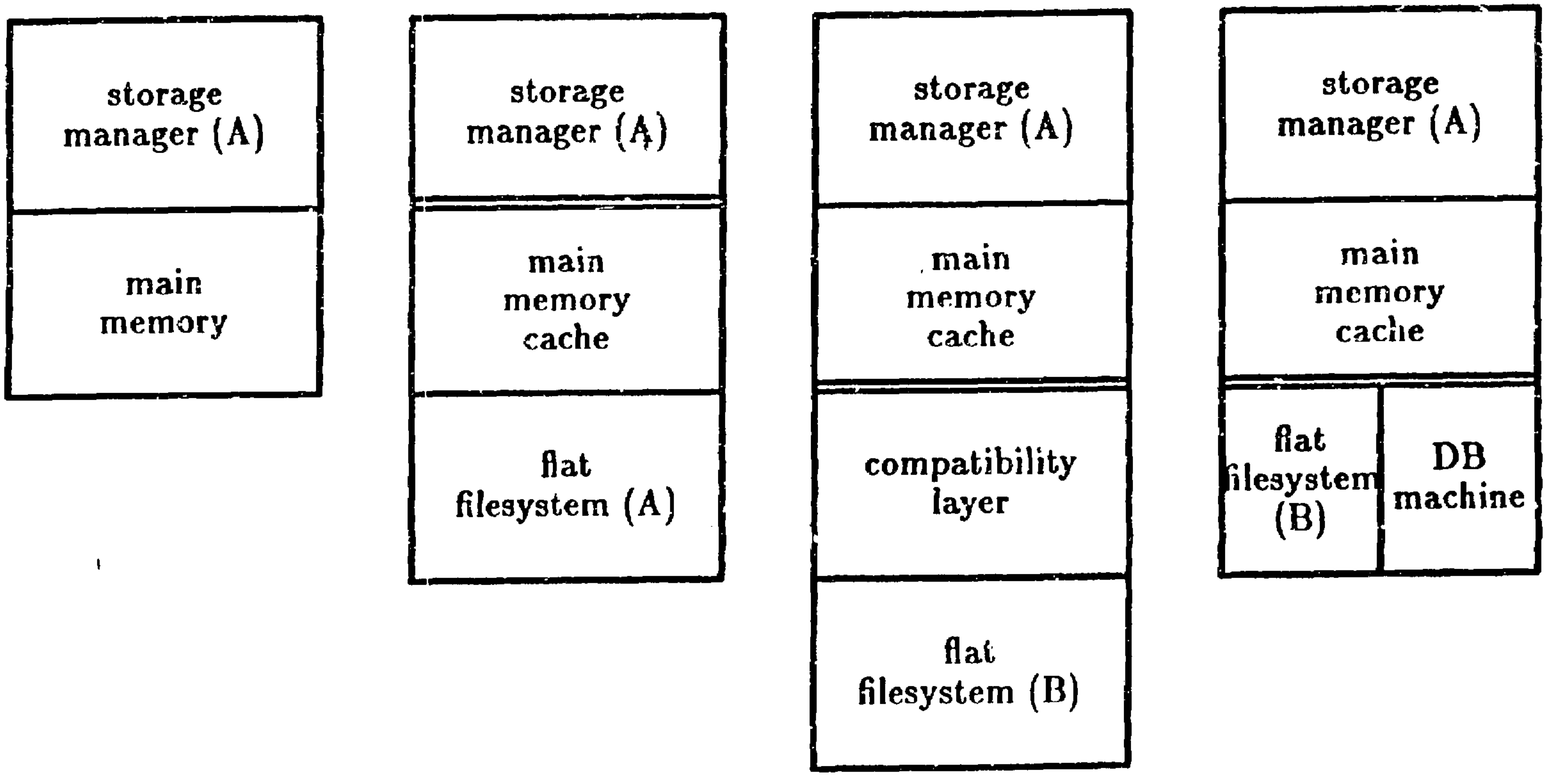
ASE

Inner Portability Layers (piggyback)



Example of Layers

- The STORAGE_MANAGER is a primitive filesystem with objects and operations suitable for building the node model.
- One or more layers must be built up from the basic storage resource level to provide this filesystem.



Abstracting Processes

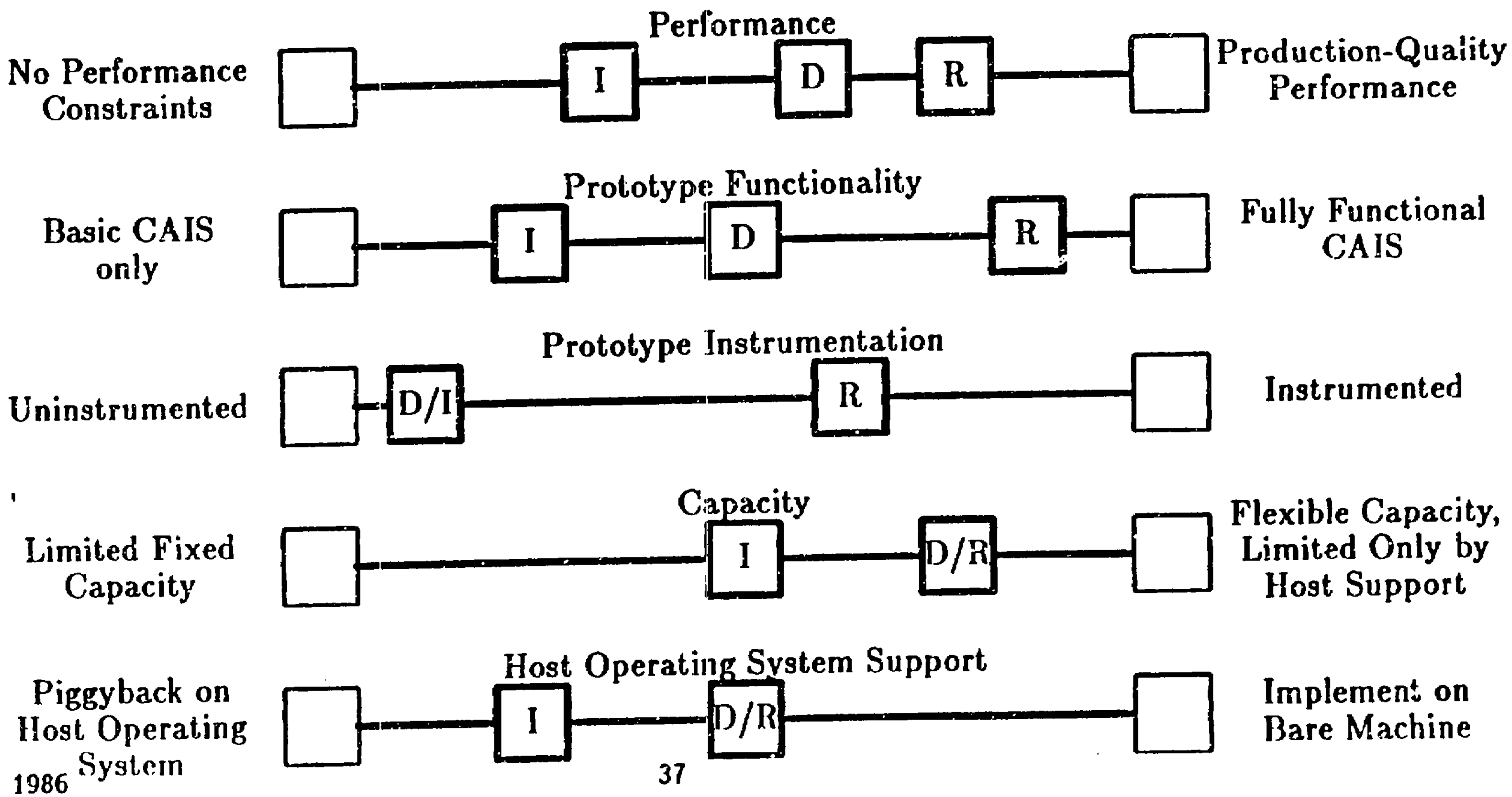
- It is much more difficult to abstract processes than storage.
- Important properties of processes:
  - How processes are created (e.g. by parent only)
  - How processes are controlled (e.g. suspended, restarted, aborted)
  - How the process's access to data is controlled
  - How the process's access to other host operations is controlled (e.g. creating other processes, communicating with other processes)
  - Which other processes are to be notified of changes in a process's state (e:g. termination with reason for termination)

Addressing the Risks

- The following slides show how the risks to the acceptance of the CAIS can be addressed by concentrating on selected prototyping objectives.
- All of the risks can be addressed in the context of either CAIS 1 or CAIS 2.
- The boxes labelled with an R show where the current prototype design needs to be extended to address each risk.

ASE | Performance | TRW

- A fairly comprehensive set of CAIS functions is needed to ensure that an efficient implementation is possible.
- Efficiency is especially important for piggyback implementations.
- Efficiency must not be achieved at the cost of severe capacity restrictions.


Performance
No Performance Constraints
I
D
R
Production-Quality Performance
Prototype Functionality
Basic CAIS only
I
D
R
Fully Functional CAIS
Prototype Instrumentation
Uninstrumented
D/I
R
Instrumented
Capacity
Limited Fixed Capacity
I
D/R
Flexible Capacity, Limited Only by Host Support
Host Operating System Support
Piggyback on Host Operating System
I
D/R
Implement on Bare Machine

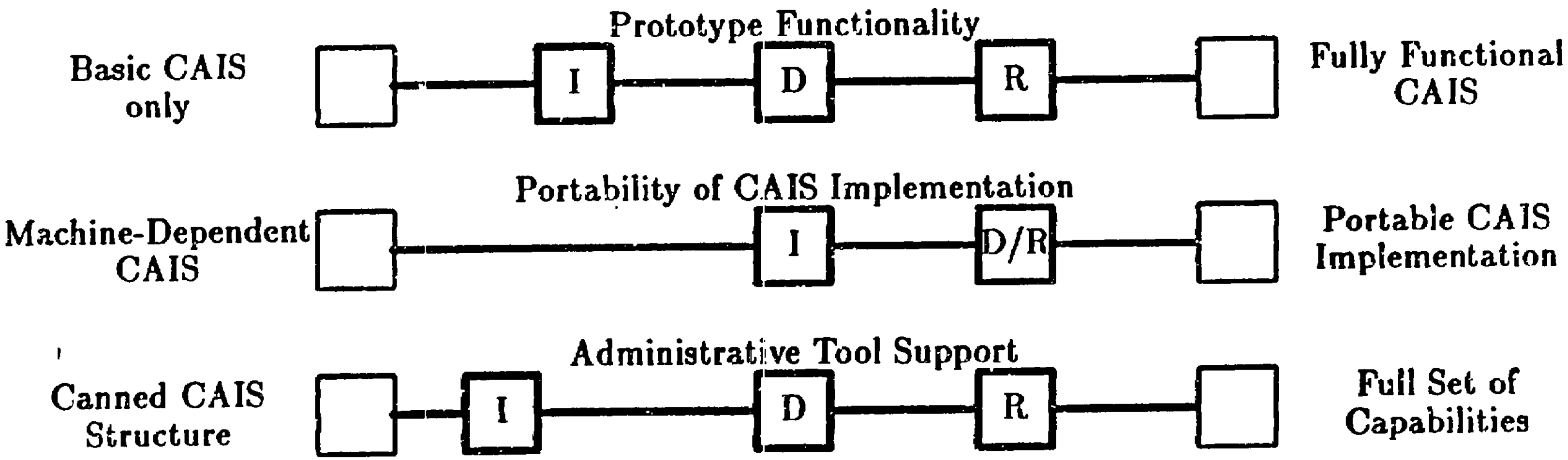
TRW's approach (Performance)

- Investigate alternative algorithms for performance-critical areas. Examples are:
  - Access control
  - Caching of disk-resident data
  - Process control
- Study the effect of various prototype architectures on performance.
  - Simulate several of these architectures.

ASE

Portability

- A fairly comprehensive set of CAIS functions is needed to ensure that all required host and/or machine interfaces are implemented.
- CAIS portability is impacted by the interfaces which support administrative tools because these interfaces may be APSE-dependent.

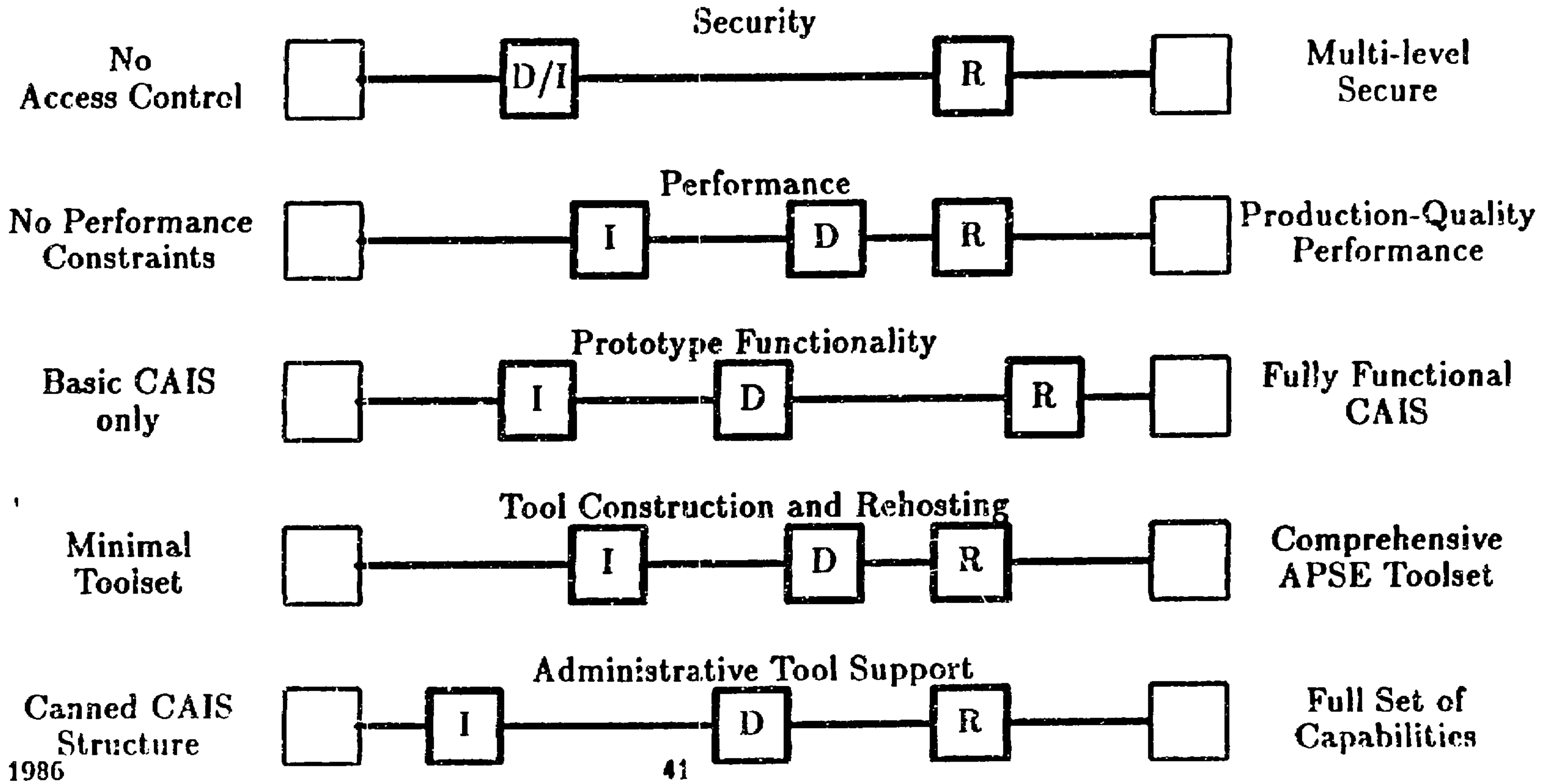


TRW's approach (Portability)

- The prototype has several inner portability layers which support the portability of various portions of the CAIS implementation.
- We are considering rehosting the prototype to at least one other host/operating system.

Security

- A CAIS implementation which can support a TCS is needed, either *as* or *on* a TCB.
- The CAIS implementation must perform adequately.
- The capabilities of the CAIS implementation must be sufficient to support all classes of tools, including those requiring administrative services.



TRW's approach (Security)

3-036

- We are investigating these CAIS development options:

  1. Building the CAIS on an "existing" TCB (eg. ASOS);
  2. Building the CAIS on a modification of an existing TCB;
  3. Building the CAIS on a new TCB;
  4. Building the CAIS *as* a new TCB.

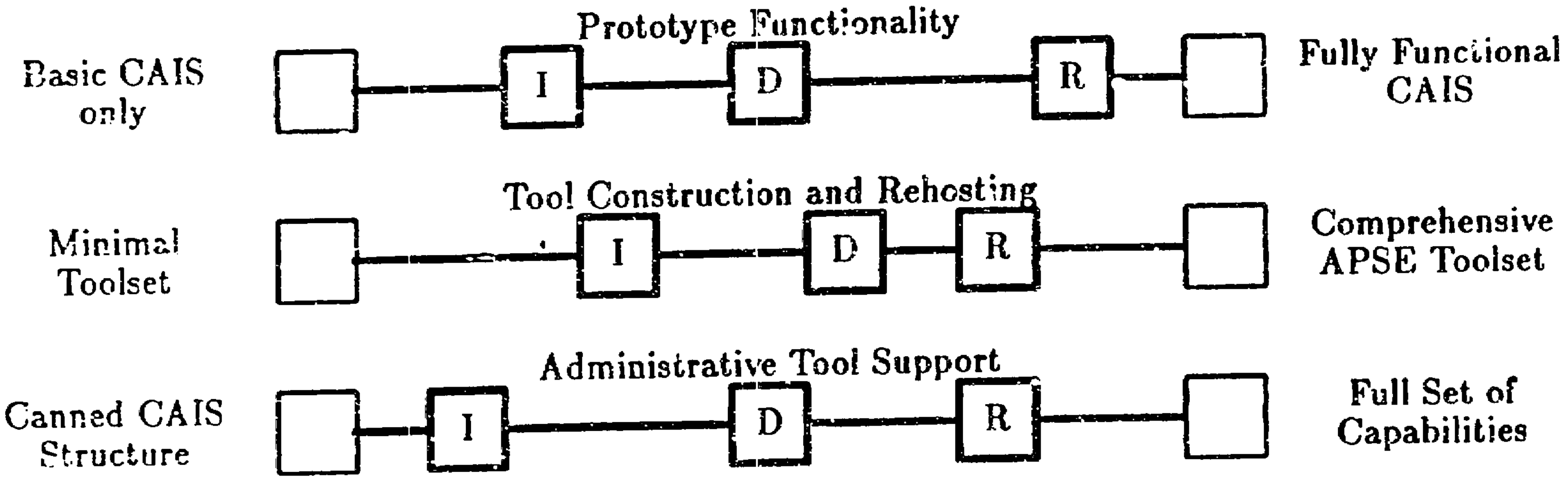
- For each of these options we are considering

  - whether the option is necessary;
  - whether the option is feasible (eg. it can be achieved with full functionality and adequate performance, and can be assured to an appropriate degreee of confidence);
  - and whether the option can be prototyped with reasonable cost.

- After the initial investigations, we will select appropriate risk-reduction approach(es)

ASE	Appropriateness	TRW

- An almost complete CAIS is needed to host a wide range of APSE tools.
- Different classes of tools which use CAIS interfaces in different ways must be considered.



ASE

TRW's Approach (Appropriateness)

TRW

- TRW plans to both port existing tools to the CAIS and to build tools on the CAIS.
- An evaluation of how well the CAIS could support an Ada implementation of TRW's Project Master Database (PMDB) is under consideration.
- We plan to design a small representative toolset which demonstrates a philosophy for constructing APSE tools. In particular, this toolset should include the browsing and access control tools mentioned earlier.

SVID As A Basis For CAIS Implementation

Herman Fischer[1]
Mark V Business Systems
16400 Ventura Boulevard
Encino, CA 91436
(818) 995-7671
{ihnp4, decvax, randvax}!hermix!fischer
HFischer@isif.arpa

1. Introduction

The Common Ada Programming Support Environment (APSE) Interface Set[2] (CAIS) is a set of interfaces, defined in Ada[3], which promote the transportability of software development tools, and which enhance the ability to move project development databases between CAIS implementations. These interfaces support large scale programming projects, such as are encountered in mission critical Defense Department computer systems work.

This paper examines CAIS as it relates to the System V Interface Definition, SVID (and UNIX[4] as a particular implementation of SVID). The paper begins by exploring why the CAIS effort exists, its goals, and the solutions it attempts to achieve which are not in today's implementations of "vanilla UNIX". Next, the paper examines the anticipated user community and why it is presumed to want CAIS. The functionalities present in the current version[2] and the functions left for later versions are identified. Two means of implementing CAIS-like functionality on host systems (such as UNIX) are identified; present implementations of CAIS are categorized and discussed. Finally, a comparison is made between CAIS and the European Portable Common Tool Interface (PCTE) project, possibly one of the most ambitious and CAIS-like UNIX extensions under way.

2. Goals of CAIS

2.1 Tool Transportability and Interoperability

The primary goal of CAIS is to solve a perceived problem in DoD: a lack of tool transportability and interoperability facilities (1) among defense system support contractors, (2) between contractors and the Government, and (3) between Government entities themselves. (The UNIX aficionado might feel that he has had the answer to these sorts of problems for years; he must be reminded, however, that neither the Government nor its contractors have historically been big fans of UNIX systems, mostly because of the inability to support programming in the *very large* on what were historically small-sized UNIX systems.)

The Government is expected to spend, this year, over $13.54 Billion on mission-critical computer software[5] (not including business and accounting applications). At typical rates of expenditures, over 126,000 software people work on over 500 defense projects (supported by many other categories of non-software labor). Several of the projects include software deliveries of the tens of millions of source lines. This code is expected to be maintained for the lengthy lifetime of military equipment; thus the ability to have many teams work on parts of the job. at differing support sites over the

[1] Mr. Fischer is chairman of the KAPSE Interface Team from Industry and Academia, and participated in the development of the CAIS.

[2] Proposed MIL-STD CAIS, *Common Ada Programming Support Environment Interface Set*, Department of Defense, Ada Joint Program Office, January 1985.

[3] Ada is a registered trademark of the U S Government, Ada Joint Program Office.

[4] UNIX is a trademark of AT&T Bell Laboratories

[5] "DoD Computing Activities and Programs, Ten Year Market Forecast Issues 1985 – 1995", Electronic Industries Association, October 1985

system lifecycle is important. (Large projects are often developed by several organizations and maintained by others. This entails a variety of computers and operating systems, and moving the project database/filesystem.)

The CAIS itself focuses on the support of software tools, in development environments. It is not intended to be a real-time or applications-supportive system, though several have suggested that CAIS facilities may apply to non-development applications too.

The CAIS currently defines an advanced filesystem (database), a process model, a security model, some device control, and some access synchronization. The difficult issue of *data interoperability* is a deferred item for the CAIS authors to tackle.

2.2 Defined in Ada

The CAIS is defined in Ada, and is intended to support tools written in the Ada language. Many of its interfaces appear in an *Ada style*, using strong typing, overloaded procedure call selection, and Ada-like packaging. There was no attempt or concern to support previous languages when CAIS was first defined; however, current interest in compatibility with other languages may influence CAIS implementations to support prior-generation languages.

2.3 Evolved from APSE Concept

In the late 1970's, Ada environment research developed the concept of a development environment architecture based on a layered model. Called the Ada Programming Support Environment (APSE), this model is shown in figure 1.

The core of the APSE is the Kernel APSE (KAPSE). It's purpose was considered novel. to encapsulate differing host machine and operating system capabilities into kernels which all had a common interface to higher level tools and user programs. The KAPSE included such general system services as file management, process control, device control, and hardware resource control.

Surrounding the KAPSE in the original models, is the Minimal APSE (MAPSE), a layer with "coding" tools such as editors,

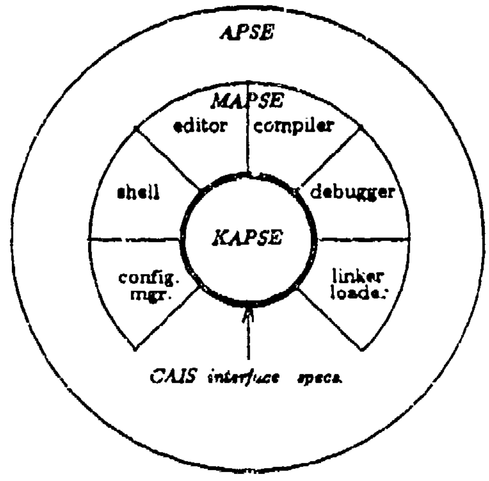


Figure 1. APSE Structure

compilers, linkers and command interpreters. Current thought focuses more on the need for an APSE to support the entire life cycle of software. This extends the support environment beyond coding tools, with such facilities as requirements analysis and design support, test support, management support, and the like. However, in the original model, these features were considered project unique tools and relegated to the APSE layer of the model.

The important contribution of this model is the idea that a kernel (KAPSE) can be defined with standardized interfaces so that low level tools (e.g., language compilers) and high level tools (e.g., project support and configuration management) can be independent of specific underlying hardware and host operating system software.

2.4 Influenced by UNIX

CAIS has been strongly influenced by UNIX. Many defense projects are still hosted on flat filesystems, such as IBM's 370 series operating systems. The CAIS designers felt the need to provide more advanced filesystem support. UNIX was seen as a model for filesystem ideas and for process control. Though UNIX was considered more advanced than many defense project environments in use, it too was perceived to have shortcomings in the area of supporting large projects. This lead to the more general *node model* in CAIS.

Fitting UNIX-like process concepts into Ada was not straightforward. Ada implies a tasking rendezvous model, which permits only synchronous parallelisms, and then only when the parallel parts are all compiled and linked together. UNIX, on the other hand, permits asynchronous parallelisms, connected by pipes and other vehicles, where each process lives in its own address space and protected from the other. Resolving these philosophical differences was not easy.

2.5 Why Ada alone (without CAIS) is not enough

With the C language, a portion of UNIX (C library) is required to augment the machine independent portion of C with sufficient functions to be useful in an operating environment. Ada must also be augmented with functions, at least in the host system environment, because it too lacks tool support functions (of the sort provided by UNIX). Two key features absent from Ada are the underlying model of the system level data, and the ability to support dynamic binding (process control). The CAIS defines a *node model* (file system model) and a dynamically bound model for multiple independent programs to inter-react as processes in real time. CAIS augments Ada with some of the (library-level) functions found in UNIX. CAIS does not define all the tool interfaces found in UNIX, and it does not define any accompanying utility programs, user shells, and the sort of functions expected of UNIX distributions.

3. Who will use CAIS

The CAIS will appeal to suppliers, customers, and projects beset with the Government's problems: namely, supporting multiple teams of software tool users on different host products, over a lengthy software system lifecycle. It is unlikely to appeal to developers of strictly single-user products (such as single-user Personal Computer software), developers of products which require hardware lock-in in order to protect their market, or developers of closed-architecture products who feel ease of integration of foreign software products erodes their market position.

3.1 Tools and tool builders

Most tool builders today strive to support a broad base of customers on a broad class of hardware. Indeed, the popularity of UNIX implementations is due to this phenomenon. CAIS carries the UNIX notion of independence further, into the Ada domain, and into a domain of a more sophisticated database capable of supporting programming in the large. CAIS may initially appeal primarily to Government contractors, and to tool builders supplying that marketplace. However, the near equivalence of non-Ada efforts, such as the European Esprit Program's PCTE project, supported by several SVID implementations for the industrial and commercial (non-Government) market, lends credence to the need for CAIS-like system functionality.

3.2 Project environments

Large programming environments need strong configuration management, the ability to support heterogeneous hosts with the same tool base, and the need to support their tool base over a lengthy time period. During the maintenance of the software, one is likely to see four or five hardware generations, and expected reprocurements of support equipment. CAIS makes Ada tools independent of hardware and underlying host OS changes.

4. What's the Present CAIS

4.1 Node model including processes

UNIX supports its users with a strictly hierarchical filesystem. For example figure 2 shows a typical user-oriented hierarchy.

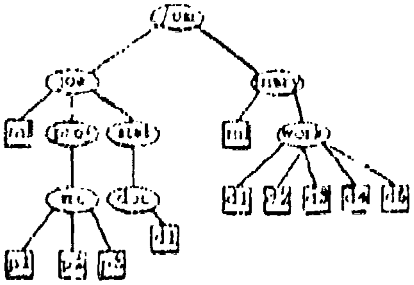


Figure 2. User hierarchy of files

Most implementations of UNIX use the directory structure to support users. Figure

2 shows two users, *mary* and *joe*, each with their independent hierarchy of files, independent of the work assignments and project considerations. For example, joe has a project directory, with a source directory for three programs (and presumably also has binary and test program directories for the same). Another hypothetical directory might include documentation. Whether mary is working on the same project or not, the files under her control would be in her own directory hierarchy.

CAIS supports building secondary networks of relationships, such as project directories with logical connection paths. Shown as curved arcs in the following figure, are a set of links from logical components to an owning project (regardless of which user owns them). Relationships can cover a number of logical connections, such as project ownerships, project version relationships, and the like (in a far more complex manner than in figure 3).

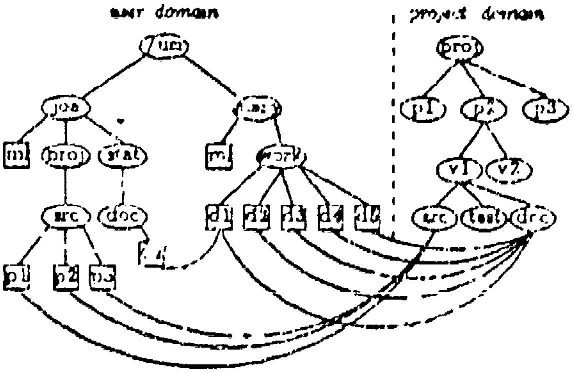


Figure 3. Network of relationships

While present UNIX distributions do not support non-hierarchical linkages or inter-filesystem linkages, some SVID extensions, such as PCTE, provide the same type of support. In general, the model underlying CAIS is one of a set of entities (e.g., tools, users, files) and their interrelationships. These may be depicted as a directed graph of nodes and edges, where the nodes represent file, device, directory, or process objects; and the edges denote relationships.

A database schema for the node model is shown in figure 4.

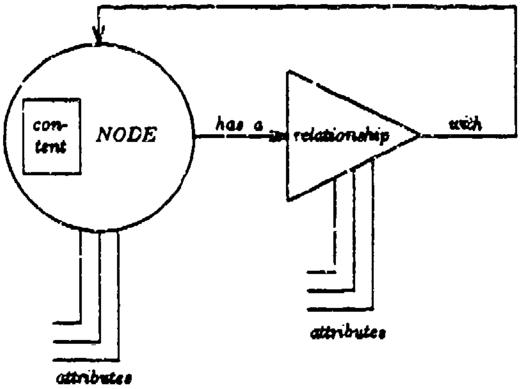


Figure 4. Database schema for CAIS node model

An important attribute of the CAIS is that processes are part of the node model. (This is similar to an enhancement to the experimental Eighth Edition of UNIX which has processes in the filesystem namespace.) With processes as named nodes in CAIS, one can have relationships between processes, between processes and file/directory nodes, and between processes and nodes for the implementation of security access models.

4.2 Terminal and Device control

CAIS defines input/output for the nodes (filesystem), as well as for terminals and tape devices. Terminal are supported as character imaging devices at present. CAIS provides support for three types of terminals: scrolling terminals, page-mode terminals, and forms-mode terminals. Scrolling terminals are basically teletype-like devices which have no cursor control. Page terminals have full screen capabilities and are equivalent to the common ANSI type of terminal (e.g., vt100). Forms terminals display fixed field menus, and receive changes to data fields, similar to some of the IBM 327x style devices

Only rudimentary device control has been provided. For tapes, the operations provided allow the CAIS to support file creation for transport between CAIS host systems. The interfaces handle labeled and unlabeled tapes.

4.3 Security Model

CAIS provides two kinds of security access control: *mandatory* and *discretionary*. Mandatory controls, equivalent to the conventional hierarchy of UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET, identify the operations of reading, writing, and reading/writing by a classifying node. Discretionary controls, equivalent to the UNIX style of user/group/other read/write/execute bits, limit the authorized access of process nodes (executing programs) (*subjects*), to other nodes (e.g., file nodes) as *objects*. Unlike UNIX, access is not controlled by storing a pattern of bits and maintaining user and group id's. Instead certain relationships are defined to other nodes to determine a node's role. Typical operations such as set-user-id are replaced by a specific process having secondary relationships such as one known as ADOPTED-ROLE.

5. What's *Not* in present CAIS

CAIS *does* provide many of the equivalent functions of SVID's system calls (UNIX manual chapter two); namely, typical kernel-level system services. In addition, some of the library functions (UNIX manual chapter three) are provided.

CAIS does *not* at present provide a number of deferred items. These include:

- Database Schema and Entity Typing methodology. Currently deferred is a decision whether or not the CAIS should enforce a particular typing methodology and what types of CAIS interfaces should be available to support it. Typing could range from simple schema representation of allowed relationships for classes of node linkages to a comprehensive control of process access to nodes depending on rules.

- Distribution. The existing definition of CAIS is intended to be implementable on a distributed set of processors, but in a manner which is transparent to CAIS interfaces.

- Advanced User Interfaces. The current CAIS does not provide interfaces for the establishment of windows or bit mapped displays.

- Inter-tool interfaces. The current CAIS does not proscribe the formats of data between tools, nor does it provide any interoperability data interfaces. The equivalent of SVID file formats (UNIX manual chapter 5) has not been determined.

- Configuration management and archiving. The current CAIS interfaces support tools which implement configuration management or archiving, but there is no proscribed underlying model for such tools to follow. In a sense this is similar to the current situation with UNIX implementations, where sites individually determine tools and procedures to follow in this regard. There is an effort under way to expand CAIS to include version control.

6. How to implement CAIS

There are two ways to provide implementations of the CAIS: a native implementation within a kernel (where the CAIS is or becomes part of the host operating system), or a *piggyback* implementation on top of a host operating system or kernel. There are prototypical examples of both forms of implementation at present.

6.1 Kernel Implementation

The only project under way which is in this category is the European implementations of PCTE, as modifications to UNIX System V.2 (see section 6.3). The implementations currently do not support Ada or Ada interfaces; however, the "C" interfaces provided will be shown to map cleanly into CAIS services. A CAIS implementation on top of PCTE would use Ada library routines, which translate the Ada interfaces of CAIS into underlying PCTE kernel services. This would not be called piggyback because the low level services in the kernel provide a significant portion of the functionality of the node model, without relying on superimposed user-state software to implement it.

6.2 Piggyback Implementation

A piggyback implementation of the CAIS might be schematically shown as in figure 5. When implemented on a UNIX environment, the CAIS implementation exists primarily as user-state coding, generally without any changes to the underlying kernel. Either shared common processes can be used for the CAIS implementation or purely user-linked coding. Two firms implementing CAIS by this technique are Mitre and Gould.

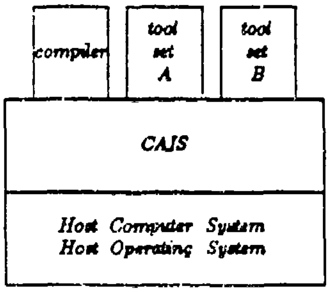


Figure 5. Piggyback CAIS implementation

6.3 PCTE

PCTE will be introduced and compared to CAIS because of two relevant points. it exists as SVID extensions, and it provides a significant part of CAIS functionality in a kernel-level implementation.

PCTE is both an interface set and a prototype implementation.

- As an interface set, PCTE exists as a set of *man* pages[6], which describe the PCTE node model, transaction processing model, distributed processing interfaces, and user interface primitives (windowing and locator device support).
- As a prototype implementation, PCTE exists as a UNIX System V kernel extension, scheduled for test in 1986. A second implementation, known as Emeraude, seeks to provide a production quality version. The PCTE prototype is part of the EEC Esprit Program, and Emeraude is a French national project.

  An additional implementation of PCTE in Ada is scheduled to be performed by Olivetti as a piggyback-styled implementation intended to be portable on a variety of hosts and processors.

6.3.1 SVID Extensions

PCTE implements a physically distributed database of objects, with a logically distributed kernel. Figure 6 shows how three workstations might share a logical distributed kernel. In this example each workstation has some portion of the database objects physically resident in its own hardware, under the control of its own local kernel, but has transparent access to all other objects of the system-wide (homogeneous) database.

In Figure 6, *UI* represents the User Interface software function of a workstation; *objects* represent database files and attributes stored locally on a workstation; and *IKC prot.* represents the inter-kernel communications protocol.

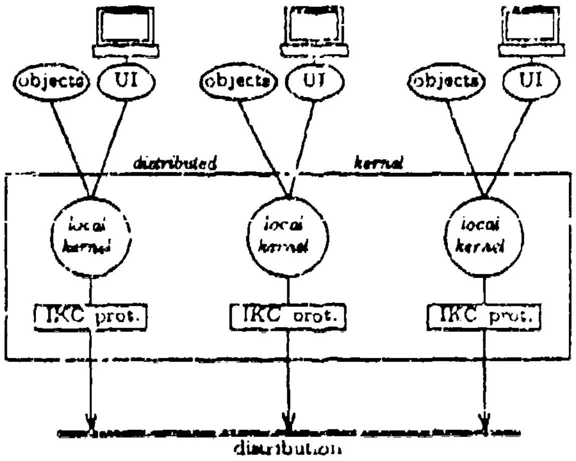


Figure 6. Distributed Kernel and Data Base

PCTE extends SVID V.2 in four logical areas. These are

1. *Basic Mechanisms.* The basic mechanisms' logical components are execution primitives, communications primitives, and inter-process communication primitives. The execution primitives, for process and context management, operate on a transparently distributed environment of heterogeneous workstations. The communication primitives provide the transparent access to distributed objects (replacing SVID filesystem primitives). The interprocess communication primitives implement piping, messages, and shared memory on a transparently distributed environment. One can start a pipeline, where pipe processes are physically separated on different

[6] *PCTE, A Basis for a Portable Common Tool Environment*, Functional Specifications, Third Edition, BULL (France) et al., 1985.

workstations, and their objects again on different workstations.

2. *Object Management System (OMS).* The OMS implements PCTE's equivalent of the CAIS node model. It is an Entity-Relationship model, based on a schema with typed nodes, attributes, and relationships (but without type-checking on process usage of OMS objects). The Schema is partitionable, so that logical views supportive of user or project needs can be implemented, and control of object relationships can be regulated. (E.g., an object program could have a derived-from relationship to a source program but not a mailbox file.) The OMS replaces the entire typical SVID filesystem, providing compatible interfaces so that binary code capability is retained for old programs ported to the PCTE implementation. It also adds support for the node model, relationships and attribute maintenance, and transparent distribution of objects.

   The PCTE OMS also provides concurrent access synchronization, both in the form of simple locking and transaction commit/abort support (e.g., rollback of object, relationship, and attribute status to state prior to commit action if a transaction sequence is aborted).

3. *Distribution.* PCTE supports fully transparent process and object distribution. It does this with only two primitives in the entire PCTE definition which explicitly reference network nodes (for explicit starting of a process on a specific workstation in the case where several may qualify for executing a certain process).

4. *User Interface.* The User Interface functions of PCTE implement a overlapped windowing system, using mouse-like locator devices, on bit-mapped terminals. The physical terminal interfaces, in one implementation, with a User Agent function, which interfaces to applications agents for each running process. Processes can either have an active window on the screen or be iconized (replaced by a symbol). The Applications Agent provides a virtual terminal for the application, so that user-state programs need not deal with window management.

6.3.2 PCTE and CAIS

PCTE is similar to CAIS in a number of areas:

- The node models are nearly identical.
- The relationship models are very similar.
- Attributes are handled in a similar manner, though schema typing in PCTE causes some practical attribute handling differences from CAIS implementations without schema support and attribute typing.
- The Process model can be installed in a similar way. Though PCTE implements processes in the manner of System V.2 (e.g., processes are identified by identification numbers which are integers), there is precedence in experimental implementations of UNIX Eighth Edition to make Processes part of the filesystem "name space". PCTE could either inherit the mechanism of that UNIX version, or it could use a library routine (outside of the kernel) to implement processes as special types of nodes.

  Ada tasks, both on PCTE and on conventional SVID implementations, are expected to be implemented by compiler libraries which place all linked tasks for a given Ada program as a single (or set of) SVID processes. In general, it is doubtful that separate tasks can be represented by independent processes; thus the process model of CAIS can be made to correspond directly to the process model of PCTE and SVID.
- Finally, Ada implemented I/O should be the same on both PCTE and CAIS implementations, because in order to validate a given compiler, one must consistently provide Ada I/O regardless of the underlying host implementation.

PCTE and CAIS differ in several areas which are important to note:

- PCTE supports a concept of schemas, subschemas, and the notion of working schemas. These can be used to restrict the logical view of objects and to control relationship and attribute mapping to objects. Nodes, classes of relationships, and attributes are typed. PCTE does not, however, perform any process to object type checking during execution (there is debate as to the implementability of process to object type checking in real time).
- PCTE supports transparently distributed processing on multiple (heterogeneous) workstations and processors sharing a Local Area Network.
- PCTE provides binary code compatibility with UNIX System V.2 tools; programs which are only obtainable in binary executable forms (and cannot be recompiled or relinked) will operate properly.
- PCTE provides a windowing user interface.
- PCTE provides a SVID-like discretionary security system, which is different from the model in CAIS.
- PCTE has no software provisions for mandatory security. The certification of the SCOMP system provides some hope that a "hardware hack" could be used to implement mandatory security for a PCTE implementation. It is also possible to use the view restrictions afforded by the Schema capabilities to implement some security functionality.

In general the primitives in PCTE can be mapped to the primitives in CAIS and vice-versa. Mappings from CAIS to PCTE are nearly complete, though CAIS lacks some of the functionality provided in PCTE. It is interesting to note, however, that the differences between Ada and SVID style impact the apparent *granularity* of primitive operations. For example, let us compare opening a node handle in the two systems:

> The CAIS call to open a node handle specifies a time limit, either a character path name or a base node and relationship from that base node, and an intent specification. These are one transaction to an Ada program (though they may represent any number of low-level operations in an implementation).

> The PCTE equivalent requires several kernel and user library-routine operations: allocating a current object (e.g., the node handle), performing an alarm(time limit), a function, chrefobj( ), to make the current object equivalent to the path to the node, and a possible lock( ) operation. These separate operations might be necessary at the "C" interface level if the "C" user wished to perform the same operations as the CAIS Ada user.

Another visible difference between CAIS and PCTE is in the handling of errors. With CAIS, the Ada style of exception raising is used, while with PCTE, the SVID style of error returns is used. Generally most implementors of Ada compilers map the error returns of SVID implementations into exception returns anyway, so this is more a difference of language usage style than an important one. There are a few ambiguities of error return to exception mappings, but these are minor.

Process control primitives differ. For example:

> In CAIS a single function call is used to spawn a process.
>
> With PCTE, the equivalent functionality would require a start( ) (of the process), a possible startact( ) (transaction locking primitive), a crobj( ) to create the node model object representation for the process node, a number of setattr( ) calls to set the attributes up for the process, and a possible lock( ) call. Of course, it is quite likely that a specific CAIS implementation would break a process spawning function call down to a number of subfunctions anyway; however, the user sees a higher level of abstraction of function call. (There is debate as to the value of abstraction granularity in this regard.)

7. Conclusion

This brief report discusses why we have CAIS, how CAIS might be and has been implemented, and how CAIS is very close to the SVID extensions now being implemented. The author strongly recommends SVID as a means of implementing CAIS.

P C T E:

"A BASIS FOR A PORTABLE COMMON TOOL ENVIRONMENT"

OVERVIEW OF PCTE BASIC MECHANISMS:

- OBJECT MANAGEMENT SYSTEM (OMS)
- EXECUTION (EXE)
- ACTIVITES (ACT)
- COMMUNICATION (COM)
- INTER PROCESS COMMUNICATION (IPC)

OBJECT MANAGEMENT SYSTEM

THE CENTRAL INFORMATION REPOSITORY SUPPORTING ALL ACTIVITIES OF PROJECTS, PROJECT TEAMS, INDIVIDUAL USERS

- TOOLS
- DOCUMENTS
- REPRESENTATIONS OF THE SYSTEM UNDER DEVELOPMENT (SOURCES, COMPILED FRAGMENTS, EXECUTABLES, CONFIGURATIONS...)
- TEST DATA
- PROJECT MANAGEMENT INFORMATION

•••

THE MAIN DESIGN OBJECTIVES OF THE OMS

- FOSTER INTEGRATION OF TOOLS
- SUPPORT ALL PROJECT STAGES, FOR SEVERAL PROJECTS
- ADAPT TO DIFFERENT METHODS AND LIFE CICLE MODELS
- SUPPORT EXISTING (UNIX) TOOLS
- SUPPORT TRANSPERENT DISTRIBUTION OF THE INFORMATION (LAN)
- INSURE THE CONSISTENCY OF THE INFORMATION REPOSITORY

SOLUTION: MERGE DATABASE TECHNOLOGY AND TRADITIONAL FILE SYSTEM CONCEPTS.

PRINCIPAL ASPECTS OF THE OMS:

- THE DATA MODEL (BASIC MODEL + TYPE STRUCTURE)
- THE WORKING SCHEMA
- THE DATA MANIPULATION FACILITIES
- THE DATA DEFINITION FACILITIES
  (OPERATIONS ON SCHEMA DEFINITION SETS)

PCTE · DATA MODEL

THE DATA MODEL

DEFINES THE NATURE OF THE PCTE'S INFORMATION REPOSITORY

IS INSPIRED FROM PREVIOUS STUDIES AND EXPERIENCES (EXPERIMENTED AVENUE, NOT JUST A JUMP IN THE DARK)

• STONEMAN, CAIS	(DoD (US))
• ALPAGE	(BULL (F))
• PAPS	(OLIVETTI, DDC, CR (CEC))
• M-CHAPSE	(BT, MoD, DoI (UK))
•••	

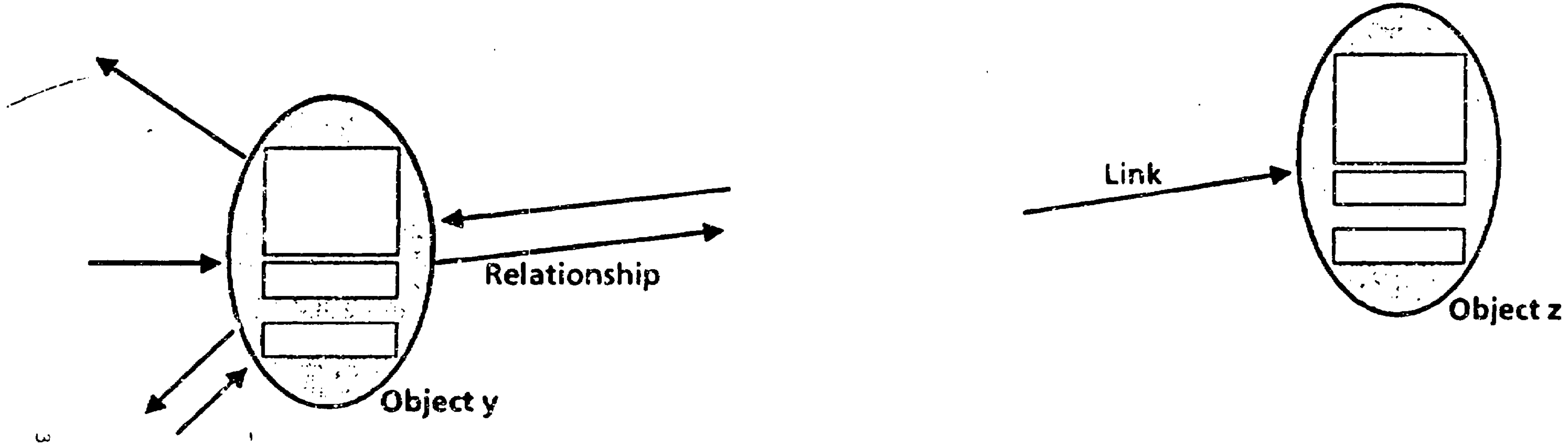
MAIN SOURCE IS THE (BINARY) ENTITY-RELATIONSHIP MODEL (CHEN)

THE BASIC NOTIONS

• OBJECTS	OBJECT TYPES
• RELATIONSHIPS	RELATIONSHIP/LINK TYPES
• ATTRIBUTES (OF OBJECTS AND LINKS)	ATTRIBUTE TYPES

Dependencies between objects

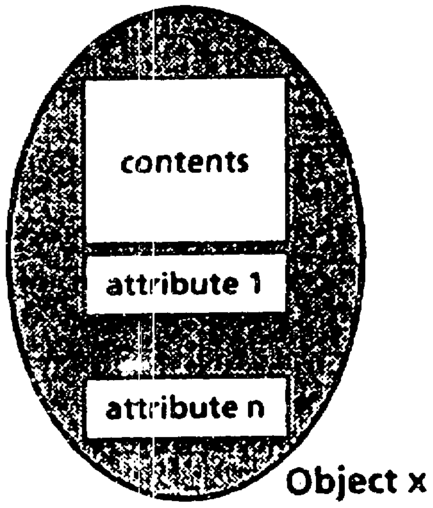
- **Links:** directed dependencies from one object to other object(s)
- **Relationships:** mutual dependency between two objects



Object Management System

Objects

- optional contents
- attributes



ATTRIBUTES

ATOMIC INFORMATION ITEMS REPRESENTING PROPERTIES OF OBJECTS AND OF RELATIONSHIP LINKS (....)

- BOOLEAN | INTEGER | DATE | STRING VALUE TYPE
- ATTRIBUTE TYPES DEFINE:
  - THE NAME BY WHICH THE ATTRIBUTE CAN BE DESIGNATED (EQUAL TO THE ATTRIBUTE TYPE NAME)
  - THE ATTRIBUTE INITIAL (DEFAULT) VALUE

OBJECT — A 'THING' WHICH CAN BE DISTINCTLY IDENTIFIED —

REPRESENTATION IN THE INFORMATION REPOSITORY OF ENTITIES RELEVANT TO SOFTWARE ENGINEERING ACTIVITIES

- OPTIONAL CONTENTS (FILES, NAMED PIPES, MESSAGE QUEUES, DEVICES)
- UNIQUE IDENTIFICATION
- OBJECTS CAN BE DESIGNATED (VIA PATHNAMES...)
- OBJECT ATTRIBUTES (PREDEFINED OR USER DEFINED...)
- BASIC MANAGEMENT OF VERSIONS (PREFERRED LINKS...)

THE CHARACTERISTICS OF EACH OBJECT ARE DEFINED BY ITS OBJECT TYPE

LINKS

UNIDIRECTIONAL, DIRECTED ASSOCIATION BETWEEN TWO OBJECTS

- SET OF ATTRIBUTES
- INTRINSIC PROPERTIES
  - SET OF KEY ATTRIBUTES (DESIGNATION OF LINKS, PATHNAMES...)
  - ARITY (TO ONE | TO MANY)
  - STABILITY
  - CATEGORY (COMPOSITION | REFERENCE | IMPLICIT)

OBJECTS CAN BE REGARDED AS NODES IN A GRAPH; LINKS ARE DIRECTED EDGES II THE GRAPH

3-678

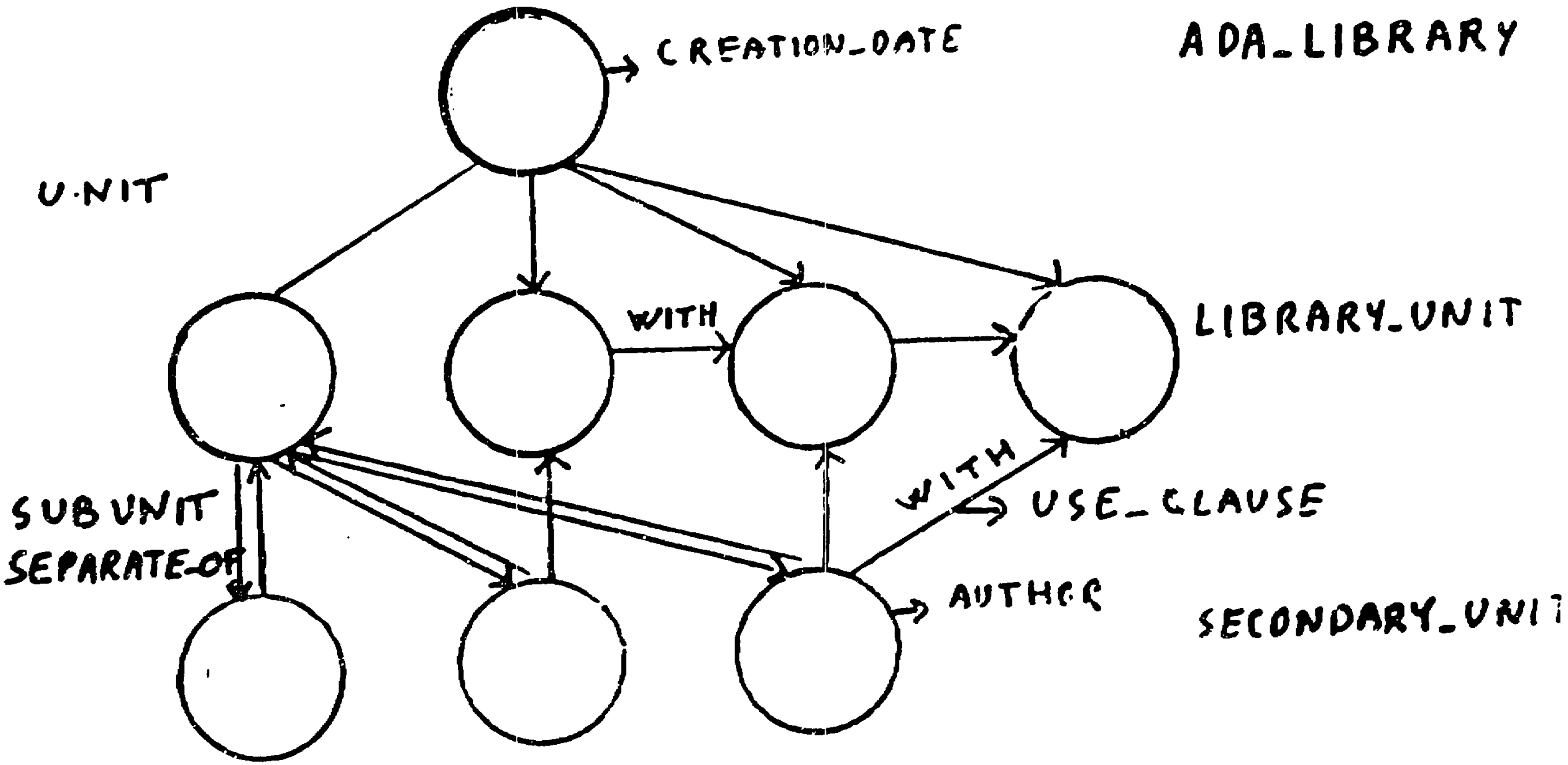
RELATIONSHIP

RELATIONSHIPS ARE BIDIRECTIONAL ASSOCIATION BETWEEN TWO OBJECTS

- TWO LINKS
  (ONE WAY, BETWEEN THE TWO OBJECTS)
- A RELATIONSHIP BINDS TOGETHER TWO LINKS
- RELATIONSHIP TYPES DEFINE CONTRAINS ON SETS OF LINKS
  - ONE TO ONE | ONE TO MANY RELATIONSHIP ARITY
  - RULES FOR WELL FORMED RELATIONSHIP TYPE DEFINITION
- RELATIONSHIP TYPES DEFINE A PARTITION ON THE SET OF LINKS FROM AN OBJECT

RELATIONSHIP TYPES DEFINE THE STRUCTURE AND PROPERTIES "IN THE LARGE" OF THE INFORMATION REPOSITORY

PCTE



Basic Mechanism

OMS

PCTE

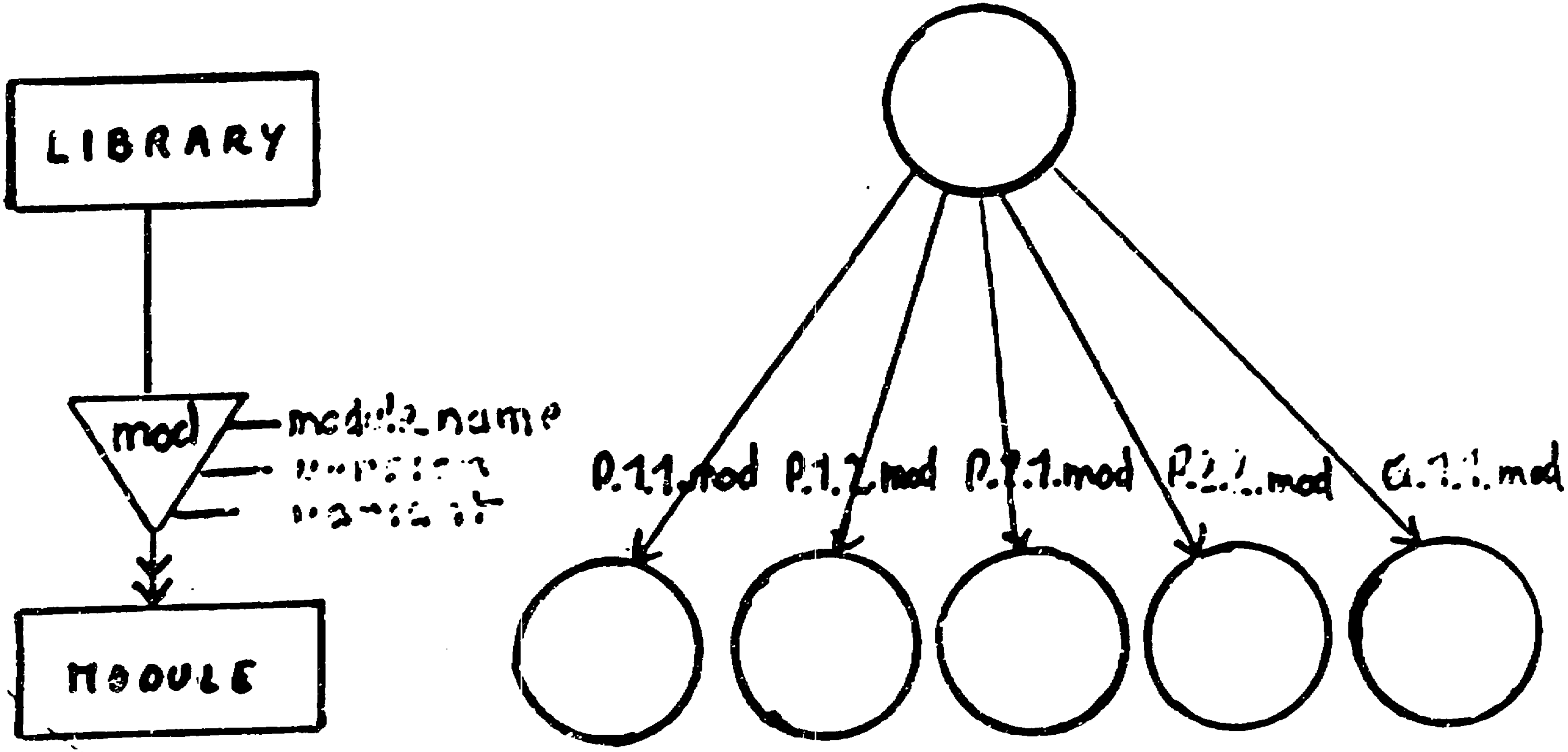
DATA MODEL

OBJECT TYPE: — MODEL FOR OBJECTS —

FOR EACH OBJECT ITS TYPE DEFINES:

- ITS BASIC NATURE: (I.E. THE PARENT OBJECT TYPE)
  FILE, PIPE, PCTE_DOCUMENT...
- ITS ATTRIBUTES
- THE RELATIONSHIPS TYPES (...) TO WHICH IT CAN PARTICIPATE
  - TYPES OF LINKS FROM THE OBJECTS
  - TYPES OF LINKS TO THE OBJECT

OBJECT TYPES DEFINE THE PROPERTIES "IN THE SMALL" OF THE INFORMATION REPOSITORIES



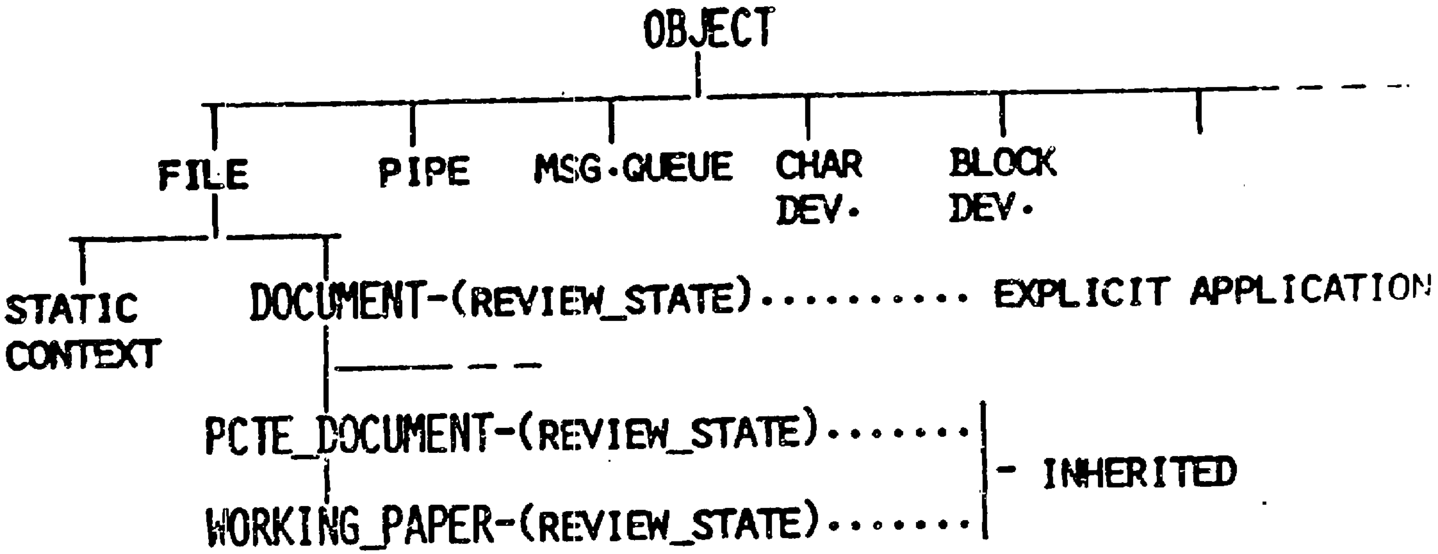
Basic Mechanism

OBJECT TYPE HIERARCHY

OBJECT TYPES ARE ORGANISED IN A HIERARCHY

- EACH OBJECT TYPE IS SUBTYPE OF ANOTHER: ITS PARENT TYPE
- TRANSITIVE INHERITANCE: DESCENDENTS OBJECT TYPES HAVE ALL PROPERTIES OF THE ANCESTOR TYPES:
  - ATTRIBUTES
  - RELATIONSHIP TYPES (TO | FROM LINK TYPES)
  - CONTENTS (EXISTENCE AND KIND)

INHERITANCE


OBJECT
FILE
PIPE
MSG·QUEUE
CHAR DEV.
BLOCK DEV.
STATIC CONTEXT
DOCUMENT-(REVIEW_STATE).......... EXPLICIT APPLICATION
PCTE_DOCUMENT-(REVIEW_STATE).......
WORKING_PAPER-(REVIEW_STATE).......
- INHERITED

SUPPORT FOR VERSION MANAGEMENT

- SIMPLE, FLEXIBLE FACILITIES
  (UNIVERSAL COMPLETE SOLUTION IS NOT FEASIBLE)

- INTEGRATED IN THE BASIC MODEL

  - PREFERRED (DEFAULT) LINK TYPE
  - PREFERRED (DEFAULT) KEY ATTRIBUTE VALUES ("-")
  - MOST RECENT KEY ("+" AND "++")

- OBJECT LEVEL (SETPREF(OMS))

- LINK DESIGNATION (PATHNAMES)

- OBJECT LEVEL AND LINK DESIGNATION LEVEL CAN BE COMBINED

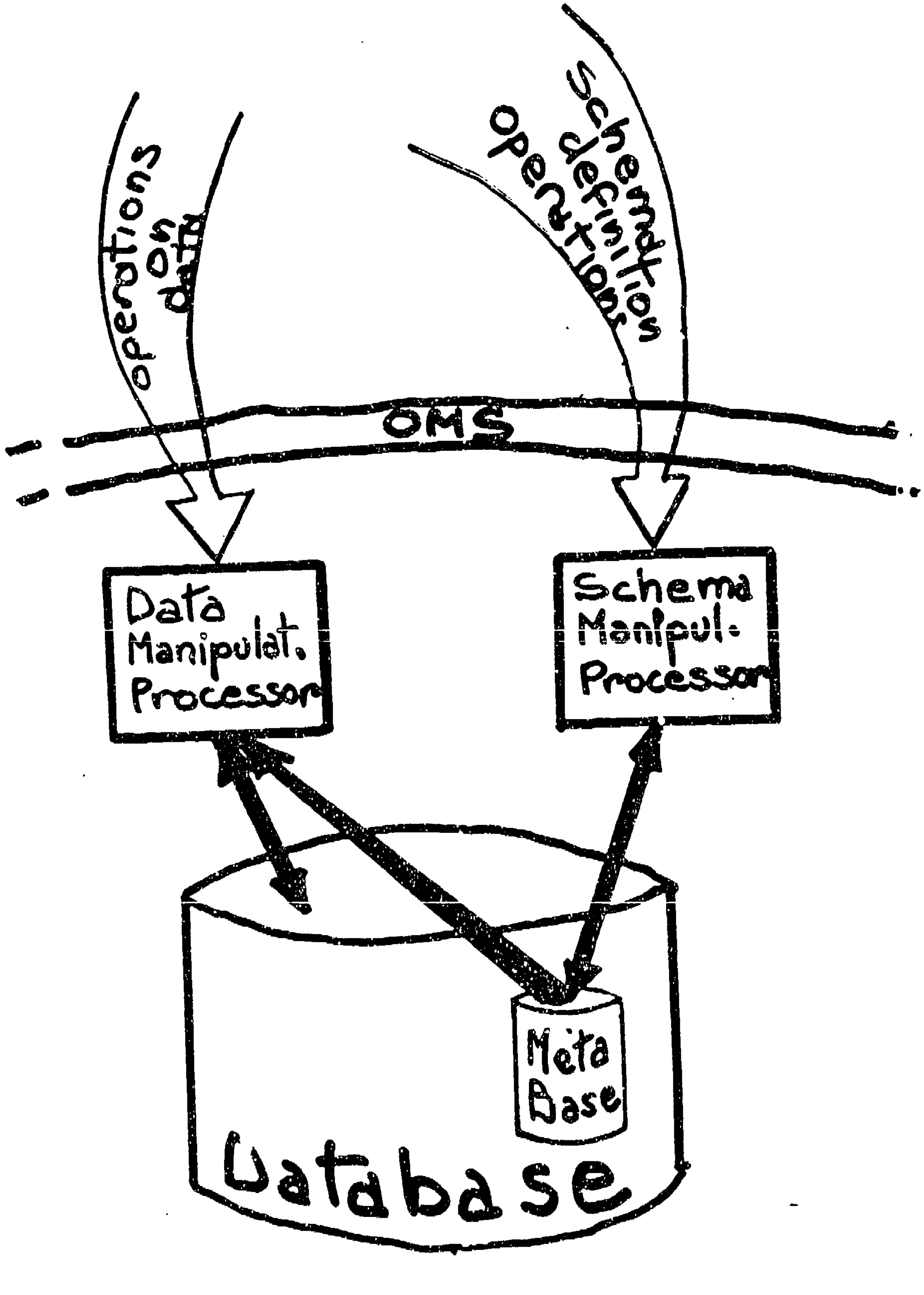
PCTE

CONCRETE SYNTAX OF PATHNAMES

```
PATHNAME          ::= REFERENCE_OBJECT ['/' RELATIVE_PATH]
                    | RELATIVE_PATH
                    | '/' RELATIVE_PATH

REFERENCE_OBJECT  ::= '/'                -- (CURRENT) ROOT
                    | '#'                -- STATIC CONTENXT
                    | '~'                -- USER 'HOME' OBJECT
                    | '.' [NUMBER]       -- CURRENT OBJECTS

RELATIVE_PATH     ::= LINK_NAME ( '/' LINK_NAME )...
```

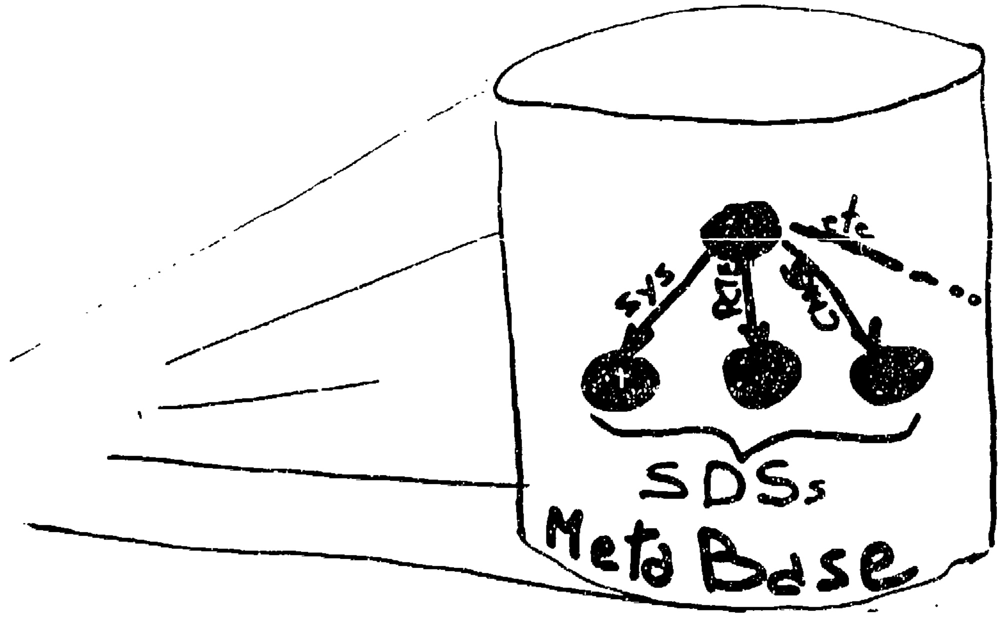
```
LINK_NAME        ::= [ KEY_VALUE ] [ '.'  LINK_TYPE_NAME  ]
                   | '.'
                   | [ NUMBER ] '..'

KEY_VALUE        ::= KEY_ATTR_VALUE ( '.' KEY_ATTR_VAL )...

KEY_ATTR_VALUE   ::= INTEGER  | STRING | '-' | '+' | '++'
```

```
/USR/BIN/ADACOMP

#/CODEGEN.TOOLGRAINS

./STACK.PUSH.ADA_LIB-WITH
```


operations on data
schema definition operations
OMS
Data Manipulat. Processor
Schema Manipul. Processor
Meta Base
Database

Conceptual Architecture

- SDSs are Objects of the OMS
  - Access Rights ...
  - Allocated into Volumes
  - Can have <u>user defined</u> links, Attributes (⇒ DD/D like Facilities)



PCTE

OPERATIONS

DATA MANIPULATION OPERATIONS

OBJECT MANIPULATION

CROBJ CRDEV CONVERT ISOFTYPE
GETTYPE MVOBJ SETPREF GETPREF
ACCESS GETOBJSTAT

LINK AND RELATIONSHIP MANIPULATION

CRLINK DLLINK GETREVLINK LSLINKS
GETLINKSTAT

ATTRIBUTE MANIPULATION

GETATTR SETATTR CHMOD CHOWN UTIME

VOLUME MANIPULATION

CRVOL MOUNT UMOUNT GETVOLSTAT

PROCESS CONTEXT MANIPULATION

CHREFOBJ(EXE) CHREFINO(EXE)
SETSCHEMA(EXE) GETSCHEMA(EXE)

PREDEFINED OBJECT ATTRIBUTES

INTEGER VALUE TYPE

OWNER GROUP MODE IVOL INO
NLINK NCLINK NRLINK NSLINK NCOMP
SIZE DEV

DATE VALUE TYPE

ADATE MDATE CDATE

INTRINSIC PROPERTIES OF OBJECTS; CAN ONLY BE CHANGED VIA SPECIFIC PRIMITIVES

WHY A SCHEMA?

TO HARMONISE TWO OTHERWISE INCOMPATIBLE REQUIREMENTS

- THE USERS (PROJECTS | PROJECT MEMBERS | TOOLS) NEED A DATA STRUCTURE ADHRENT TO THEIR EVOLVING NECESSITIES
- THE SYSTEM MUST BE FLEXIBLE, SUPPORT DIFFERENT NECESSITIES
  - CANNOT FORCE A RIGID, STANDARD MODEL
  - CANNOT SUPPLY A STRUCTURE-LESS "SPAGHETTI BOWL" MODEL

A SCHEMA IS A PARAMETER TO THE SYSTEM TO SPECIALISE A FLEXIBLE DATA MODEL

WHAT A SCHEMA IS FOR?

- UNIFORM CRITERIA TO ORGANISE THE INFORMATION REPOSITORY
  (CATEGORISATION OF INFORMATION ITEMS)
- LOGICAL CONSISTENCY RULES
- DEFAULTS AND IMPLICIT SEMANTICS WHICH CAN BE INFERED BY OMS DATA MANIPULATION OPERATIONS

THE OMS TYPED DATA MODEL INCREASES RELIABILITY AND

- CAN REDUCE COMPLEXITY OF TOOLS
- CAN AUGMENT EFFICIENCY OF TOOLS
  (THE SYSTEM CAN CHECK CONSISTENCY FASTER)

WHY SEVERAL SCHEMAS?

- DECENTRALISE THE MAINTENANCE OF THE TYPE STRUCTURE REPOSITORY (NO DATABASE ADMINISTRATOR ROLE)
  - PROJECT TEAM LEVEL RESPONSIBILITY
  - INDIVIDUAL USER LEVEL PERSONAL WORKING STYLES

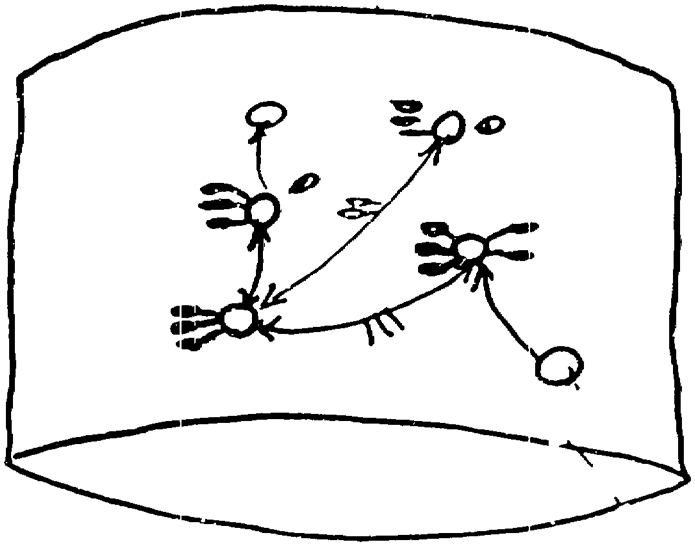
  SCHEMAS ARE EVOLVING THINGS

- ACTUAL SUPPORT FOR TOOL TRANSPORTABILITY
- ACTUAL SUPPORT FOR TOOL ACQUISITION, INTEGRATION

A SINGLE SCHEMA CAN BE A SERIOUS OBSTACLE TO DISSEMINATION OF TOOLS

CMS
Managem.
tool
Browser
(ls)
Compiler

Basic Mechanism



STRUCTURE OF THE METABASE

- THE META BASE CONTAINS THE TOTALITY OF THE OMS TYPE DEFINITIONS (THE "LOGICAL SCHEMA")
- A SCHEMA DEFINITION SET (SDS) IS A PROJECTION OF A FRAGMENT OF THE METABASE (A "SUBSCHEMA")
  - PARTIAL, SELF-CONTAINED VIEW OF TYPE DEFINITIONS
  - POSSIBILE RENAMING OF OBJECT, LINK, ATTRIBUTE TYPES
  - SDSs MAY HAVE INTERSECTIONS WITH OTHER SDSs

PCTE DATA DEFINITION

OPERATIONS ON THE METABASE (SDS OPERATIONS)

- DATA DEFINITION FACILITIES OPERATE ON SDSs
  (THE "MAPPING" TO THE METABASE IS AUTOMATIC)
- CREATION | MODIFICATION | DELETION OF TYPE DEFINITIONS CAN BE DONE AT ANY TIME BY ANY (AUTHORISED) USER
- SDS ARE OMS OBJECTS (OF PREDEFINED TYPE)
  - ACCESS RIGHTS
  - THE METABASE IS DISTRIBUTED
  - USER DEFINED SDS LINKS | ATTRIBUTES
    (DD/D LIKE FACILITIES...)

DATA DEFINITION OPERATIONS

SDS LEVEL

SDSINIT SDSOPEN SDSCLOSE SDSDEL
SDSDESNAME SDSNAME

DEFINITION LEVEL

SDSCRDEF SDSIMPDEF SDSEXTDEF SDSAPPPLY
SDSNAMEDEF SDSSETINFO

REVOKING

SDSDLDEF SDSRVKDEST SDSUNAPPLY

BROWSING

SDSSCAN SDSGETDEF SDSDEFNAME SDSDEFID
SDSDEFSTAT SDSTYPREF

WORKING SCHEMA

PROCESS LEVEL VIEW OF THE TYPE STRUCTURE

- "WELL FORMED COMPOSITION" OF AN OREDERED LIST OF SDSs
  - UNION OF THE TYPE DEFINITIONS IN EACH SDS
  - PRECEDENCE RULES TO SOLVE CLASHES
- THE WORKING SCHEMA IS PER PROCESS
  - ESTABLISHED BY THE PROCESS ITSELF (SETSCHEMA(EXE))
  - IS INHERITED BY CHILD PROCESSES
    (CAN BE CHANGED IF NEEDED)

TYPE STRUCTURE

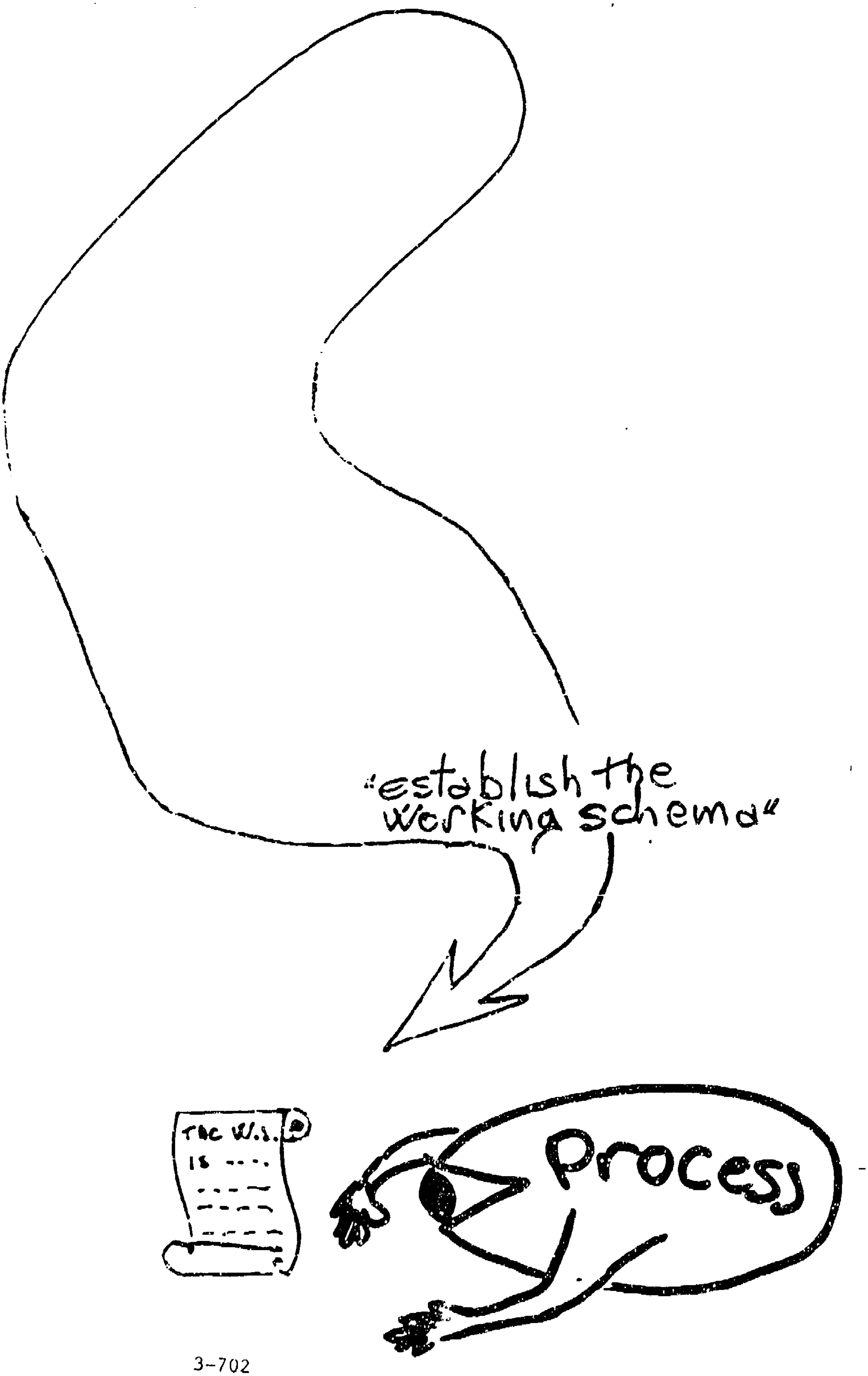
EXPLOITATION
ALL ACCESSES TO THE OMS ARE RULED BY PER PROCESS CURRENT WORKING SCHEMAS

CONSTRUCTION
DATA DEFINITION FACILITIES ALLOW THE IPLEMENTATION AND THE MAINTENANCE OF THE OMS TYPE STRUCTURE (SDS*(OMS))

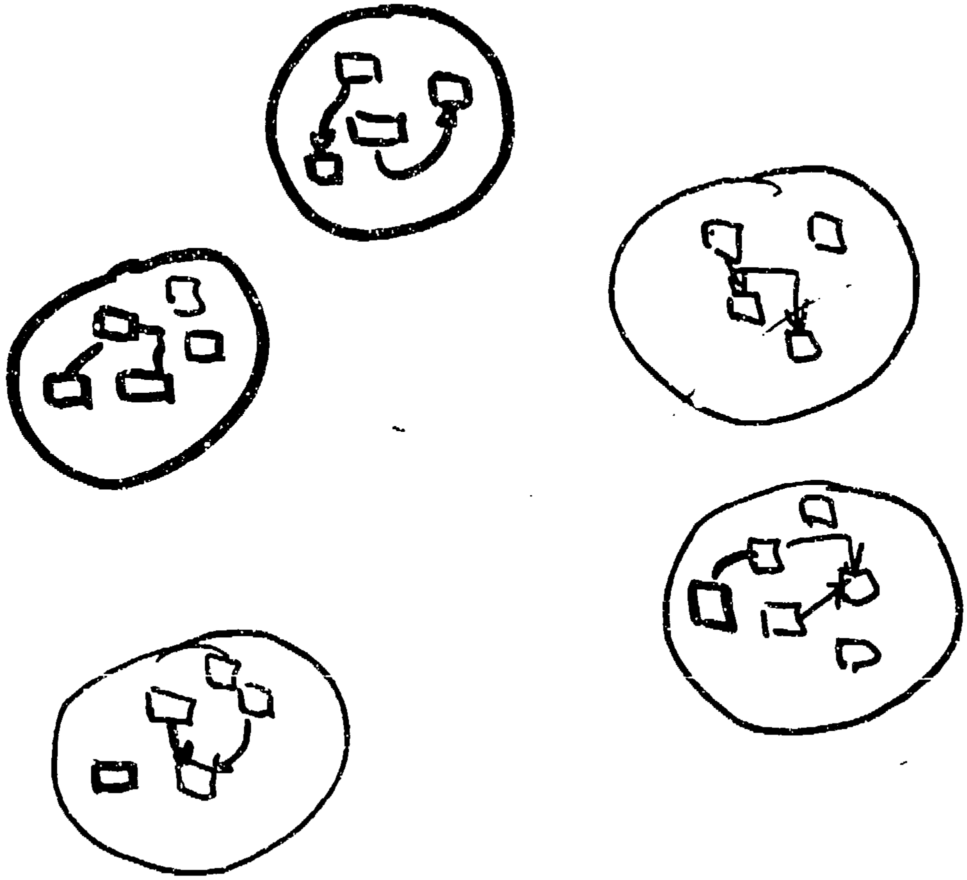
FLEXIBLE MODEL FOR DECENTRALISED, MODULAR CONSTRUCTION AND EXPLOITATION OF SOFTWARE ENGINEERING DATABASES

SUMMARY OF SCHEMA MECHANISMS

- DYNAMIC PROCESS LEVEL WORKING SCHEMA ("SUB-SCHEMAS") FROM A LIST OF SCHEMA DEFINITION SETS
- SDS ARE SELF-CONTAINED FRAGMENTS OF THE METABASE
    - ALIASING
    - PROJECTION OF TYPE DEFINITIONS (GRANULARITY, MODULARITY)
- SHARED AND PRIVATE SET OF DEFINITIONS
- SDS CAN BE MODIFIED TO MAINTAIN THE TYPE STRUCTURE (SDS*(OMS) OPERATIONS)
- NO NEED FOR A DATABASE ADMINISTRATOR
- SDSs ARE OMS OBJECTS

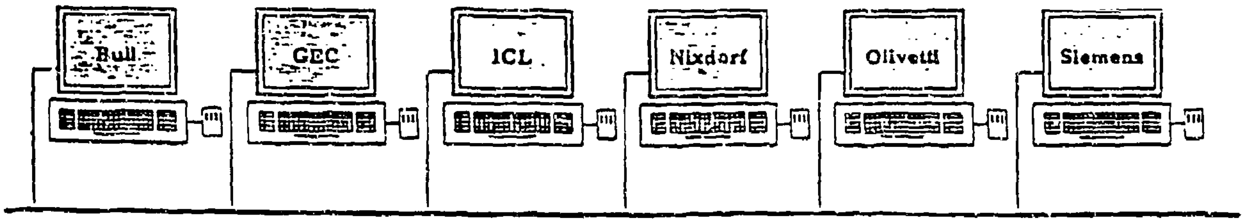

"establish the working schema"
THE W.S. IS
Process

SDS are subsets of the Meta base



PCTE

A Basis for a Portable Common Tool Environment



ESPRIT TECHNICAL WEEK 198

A Basis for a Portable Common Tool Environment (PCTE) Design guidelines

PCTE project team+

This paper outlines the organisation the PCTE project and describes the fundamental guidelines which inspired the PCTE design efforts. Finally, it also presents the main aspects and the rationale for the PCTE implementation strategy.

1. The PCTE project

The project "A Basis for a Portable Common Tool Environment (PCTE)" is carried out by a consortium led by Bull (France) and including GEC and ICL (United Kingdom), Nixdorf and Siemens (Federal Republic of Germany) and Olivetti (Italy). The purpose of the project is to design and implement a software system to serve as basis for the development of complete, modern Software Engineering Environments.

The project started at the end of of 1983 and is to run for a period of four years. Within the Consortium, Bull, ICL and Siemens jointly develop the UNIX* based PCTE version; Olivetti is responsible for the early implementation of a PCTE prototype and for a longer term Ada** version of PCTE; GEC and Nixdorf develop two sample tools: the Knowledge Based Programmer Assistant and the Configuration Management System. These tools will exercise and demonstrate the validity of the PCTE design.

The project milestones and deliverables are defined so that intermediate results from the project can become visible and available to the ESPRIT community in time to be the basis for the integration and the dissemination of the results of other R&D projects. The first result of the project has been the production of the PCTE Functional Specification Report. The F.S. report gives the detailed definition of the PCTE functionalities in a form which can directly be used in the design of tools and programs which will eventually be integrated into the PCTE hosting framework.

PCTE will be developed in close cooperation with other ESPRIT projects, in particular with GRASPIN (graphical specification and formal implementation of non-sequential Systems), SPMMS (Software Production and Maintenance Management System) and the ROSE infrastructure project.

This paper concentrates on the discussion of the overall PCTE design guidelines; the actual functionalities of PCTE are described in the PCTE Functional Specification report (the third edition of which is publicly available inside ESPRIT).

+ contact person: Mr. J.P. Bourguignon
PCTE programme manager
Bull - DRTG/GL - 58F23
68, Route de Versailles
78430 Louveciennes
FRANCE

* UNIX is a trademark of AT&T Bell Laboratories

** Ada is a trademark of the Ada Joint Program Office

2. Introduction

The area of Software Technology is one of the five areas of the ESPRIT programme which are recognised as prioritary in order to preserve and improve the competitiveress of the European Information Technology Industry. Projects undertaken in the S.T. as well as in the other ESPRIT areas will require software to be developed and exchanged by various teams all over the Community. This aspect has led to the notion of a common environment, to be used by the different research teams involved in ESPRIT, and facilitating not only the development, but also the exchange of software between the teams. Such an environment should provide a suitable basis for the various projects, and be available on different categories of machines, so as to cover the needs of the various teams.

A suitable Software Engineering Environment (SEE) should offer specific facilities that are generally not found in conventional systems. The natural conclusion is therefore to use a dedicated system for development activities, distinct from the machines for which the software is developed. This host/target development paradigm is generally admitted in the context of embedded systems, but appears more and more necessary also in the context of general data-processing systems, in which the specific requirements of development activities are often seen as nuisances.

Software development is accomplished more and more with the new generation of personal workstations linked together by fast Local Area Networks. These workstations are personal computers with pointing devices and high resolution raster displays capable of supporting several character sets and graphics; the LAN facilities can provide each workstation with several services, such as file and data base services, print services, software development services, and communication services, both store-and-forward (e.g. electronic mail) and dialogue (conversational). The main advantage of these kinds of environments is that they put the power of a time-sharing system in an immediate and direct fashion into the hands of one user, allowing the user to carry on several different or related activities as they best suit his needs and preferences.

The PCTE takes into account the current evolution towards advanced workstations and the use of local area networks, and defines basic concepts for Software Engineering Environments which can be adapted for both conventional and distributed systems. Thus, the distribution of functionalities is an important factor, and the human interfaces are oriented towards the best use of modern technology. However, because PCTE should be usable in realworld contexts, where conventional hardware (main frames and CRTs) will remain installed, the environment should also operate in such contexts. Of course, information will be available in a less comfortable fashion on conventional systems.

Most software engineering tools that are presently available result from individual efforts and tend to constitute a collection of vaguely related products, each filling a particular function, but without much consideration for the software development process as a whole. On the other hand, research in software technology will lead to the implementation of a variety of integrated tool sets to support theories, methods, and production of software. These tools will most likely be developed in the context of different projects. It is therefore important that PCTE offers a structure based on state-of-the-art technology that can:

- reduce the development costs of software tools, contributing to their widespread use, and therefore, to an improvement of software productivity.

- facilitate the exchange of software (tools and products based on the PCTE public common tool interface) among the European software community.

- allow for and encourage the integration of tools in comprehensive, uniform and homogeneous Software Engineering Environments;

- speed up the dissemination of ideas and techniques among and between the industrial and the research communities by providing a common frame of reference.

- support the smooth transition from existing practices by allowing the easy migration into PCTE of existing tools.

These considerations naturally lead to the concept of a unifying framework that could be used for the development and integration of a variety of tools in order to constitute a family of complete environments, each one with its own specific characteristics in terms of methods, application area, etc.

The PCTE project is aimed to the definition and the implementaion of a common framework, within which various software tools can be developed, integrated and exchanged so as to provide complete environments for software engineering. Different trends in software technology are focused in PCTE:

- the openness of an environment, encouraging people to develop their own tools as epitomised by UNIX;

- the need to conceive an environment around powerful mechanisms, especially in the area of object management, corresponding to the Stoneman philosophy;

- the improvement of programmers' productivity by means of powerful user interfaces, an avenue explored especially in the Interlisp, Mesa, Smalltalk, and Cedar environments developed at Xerox PARC.

As common framework for advanced software engineering environments, PCTE can have a significant impact on the European Software Technology industry; however, the objectives of the PCTE initiative can only be achieved if PCTE will become available sufficiently early to be used effectively within ESPRIT projects and if it can gain sufficient acceptance inside ESPRIT, as well as inside the industrial community at large, to demonstrate its suitabilty as de facto standard.

Thus, the PCTE main objective is to provide a powerful, state-of-the-art system that offers high-level mechanisms for the development and integration of a variety of software tools. However, the success of the PCTE initiative also relies on the achievement of two main strategic goals:

- Portability: PCTE shall provide an environment that can be made available quickly on a wide scale without incurring large costs.

- Compatibility: PCTE shall allow a smooth transition from existing software development practices.

3. PCTE preferred architecture

A state-of-the-art development environment has to provide above all a great comfort of use. Such comfort can be characterised by:

- the raw power available to the user in terms of instant computing power and large storage capacity;

- the quality of the dialogues with the machine: one of the key elements there is the so-called bandwidth of the flow of information, whereby for a given effort from the user, one can increase the amount of information entered or displayed;

- the overall ease-of-use of the system;

- the general availability of the facility.

These considerations quickly led to the conclusion that, although the system should be available on a large variety of machines and architectures, there was an optimal choice on which we should focus our efforts.

The preferred architecture would consist of powerful, single-user workstations offering a reasonable amount of local computing power (0.5 - 1 Mips) and main storage (at least 2 Mbytes). These workstations should be equipped with a high-resolution display, capable of the textual and graphic representation of large quantities of information (corresponding to several A4 pages), and some pointing device.

However, it is important to realise that the development of a large or complex piece of software results above all from the work of a team. This should be reflected in the environment, which should be an environment for a project, and not only for individuals. The various workstations must therefore be connected physically so that a user working on one workstation can access information located on, or communicate with, other workstations. They must also be connected logically so as to treat a user as an actor in a larger context, rather than as an individual who interacts with other individuals.

The architectural support can thus be seen as consisting of a set of workstations connected through a high-bandwidth local-area network and sharing common physical and logical resources ("servers") such as printers, disks, communication channels or specialised processors.

The software architecture should however preserve the visibility of a unique set of resources accessible to everyone. In other terms, we should have a single system, whose resources are distributed in a transparent manner among the various physical components, and not a number of individual systems that have to communicate explicitly. This insistence on the transparency of the distribution is seen as fundamental if we want to preserve the portability of the system towards different kinds of hosts.

4. PCTE design objectives

We detail below the principal individual objectives that we strived for during the design of PCTE facilities, and how these can be interpreted in the context of the resulting functionalities.

4.1. Generality

The hosting structure should be capable of providing the basis for a number of environments, differing in such aspects as the application domains, the development methods, the project organisations, or the programming languages used.

The basic environment, on the other hand, provides powerful mechanisms upon which these specialised environments can be built and operated. The mechanisms should therefore cater to a variety of needs, offering the maximum relief to the tool developer, while avoiding any interference with his design decisions.

Of particular importance in this respect is the need to separate the mechanisms from the policies that can govern their use. The basic environment should offer ways to control the use of the various primitives, but should not impose any control of its own.

4.2. Flexibility

A consequence of the approach described above is that the structure must be adapted to a variety of needs. This tailoring can be achieved effectively if the basic primitives offered are themselves fairly simple, but represent a complete set of functionalities. Thus, it is always possible to build more complex functions on top of the basic ones, thereby tailoring the upper layer that will be made visible to the end user.

It is important not to be preemptive in this respect, and to recognise the existence of three categories of actors dealing with the environment:

- The tool developers construct tools and software layers as to tailor the environment to specific needs.

- The SEE architect constructs an integrated project support environment with the PCTE and the appropriately chosen tools.

- The end users merely use an environment together with its tools in the context of their daily work.

The design has to cater for the needs of tool developers, SEE architect and end users; the three categories represent the "users" (in a general sense) of the system.

4.3. Homogeneity

One of the most salient demands placed on an environment from the user's point of view is the homogeneity among a given set of tools, as well as of the environment as a whole. Homogeneity appears at three different levels:

- A logical level which corresponds to the functions performed by the various tools: each tool should have a precise function that complements exactly those of other tools, without overlapping with them. Within the context of a particular development approach, each tool has its place, and the whole set of tools should provide complete support throughout development and operational use.

- An internal level corresponding to the objects and operations accessed by the tool. These are governed by the facilities offered by the basic system, but also by design choices regarding the various tools, such as the internal representations of the manipulated information which must be consistent between the various tools.

- An external level which defines the interface between the end user and the tool. It is important that the man-machine interfaces for the various tools be uniform and consistent.

The logical level is mainly dependent on the choice of a particular development approach and of the associated tools. The internal level is also largely dependent on implementation choices. However, the basic environment can indeed foster the homogeneity of operations and the integration among the various tools by offering an adequate set of functionalities, well adapted to the particular needs of tool development. The notion of Software Engineering Database is clearly central to this issue; therefore, the Object Management System is considered as a central feature of PCTE.

At the external level, much can be done to promote homogeneity as perceived by the end-user: the forms of dialogue, the style of command languages, the provisions for on-line assistance, can be defined at the level of an entire environment, and supported by mechanisms and meta-tools that can facilitate their adaptation to a particular, given tool.

4.4. Portability

Two main phases and two complementary approaches are identified for the implementation of PCTE basic mechanisms:

- A medium term phase, centred around portability of tools which stresses compatibility with UNIX as the means to exploit and reuse the large pool of existing UNIX tools.

- A longer term phase, centered around Ada and a portable architecture to obtain the highest degree of portability of PCTE basic mechanisms.

4.4.1. The UNIX based phase

The technical approach to portability reflects the will to achieve a widespread availability of the PCTE both in a short time frame, and in a cost-effective manner. Due to the portability of UNIX and its wide availability the development is based on extensions to UNIX. This is the best compromise to transport the PCTE onto a wide variety of machines at a reasonable cost and in a short time-scale.

The requirements for compatibilty with the existing practices and for a quick, widespread availability has led us to decide on a UNIX-based implementation strategy. However UNIX is far from being a distributed system. Trying to rewrite the UNIX kernel to fit on a distributed architecture would go against our goal of being able to rehost the environment on existing UNIX systems.

So as to assure portability, a layer of communication facilities will be added onto UNIX so as to provide a base of communicating nodes, each one running a separate UNIX system. The mechanisms and functions of the environment itself will be implemented as one global system on top of the collection of individual UNIX systems.

4.4.2. The Ada based phase

The above analysis concerns machines for which UNIX is already available (i.e., we do not incur the cost of transporting UNIX itself). On the other hand, should UNIX have to be transported as well, then the approach might be reconsidered. This is the basis for undertaking a complementary implementation of the PCTE fully written in Ada, in top of a minimal portability kernel.

An alternative implementation in Ada of the basic mechanisms is thus the long term goal of the project. The approach stresses the portability of PCTE basic mechanisms and thus of SEEs built on top of it on systems which are not constrained to the UNIX family. A key issue is the definition of an internal portability level which can be easily mapped on top of a variety of Operating Systems.

The effectiveness of this alternative stems from the belief that, in the long run, Ada compilers will be available for a large number of machines. The choice of Ada is based on the two following considerations:

- Ada is a modern language specifically designed for writing portable and reliable system software

- Ada is expected to become widely accepted in the R&D community.

The approach is also justified by the fact that market pressures will cause Ada compilers to be developed, independently of the need for the PCTE.

This dual (UNIX and Ada) approach is aimed at providing the best portability within different time frames; these considerations should not have an impact on the users of the environment, who should see the same set of primitives, the PCTE common tool interface, and use them in exactly the same way. Thus, the tools developed for the UNIX based PCTE will still be portable to the Ada based implementation, which, according to the multilanguage nature of PCTE, will also offer consistent C language interfaces.

4.5. UNIX compatibility

The support for migration path from UNIX based environments to PCTE is a principal short term goal of the project. This is a vital factor which will determine the future role of PCTE. In this connection PCTE has, first of all to gain acceptance as a ready available, easy to adopt alternative to existing practices; it must be possible for tools, data and people (project teams or individual users) to migrate towards the new environment with a minimum of efforts. The key issue in this respect is the ability to directly reuse existing tools, thus the absolute requirement for a complete compatibility of PCTE facilities with the corresponding UNIX ones.

5. Implementation strategies

5.1. Exodus - moving to PCTE

A reference scenario is envisaged which describes the migration of projects from an existing environment to PCTE. The preferred situation assumes this environment being based on UNIX System V running on either some (LAN of) workstations or on a minicomputer. Moving from UNIX based SEEs to PCTE can actually be carried out simply by exploiting the up-ward compatibility of PCTE with UNIX. On the other hand, PCTE offers much more powerful capabilities and further evolution possibilities which can be taken into account from the initial phases of the migration process.

The migration of a project from one SEE to an other implies the transport of the tools and data structure supporting project team activities and to reproduce on the new system the characteristics of the operational environment the project team members are familiar with. The most obvious aspect is the emulation of the CRT style of User <=> System interaction protocol: though the adaptation of the user to the more advanced facilities of PCTE User Interface (windowing, pointing devices, menus) can be achieved at no cost, environment users should be allowed to reuse, at least in the initial steps toward the new facilities, features such as shell scripts, aliasing and profiling. Thus the UNIX shell has to be made available to make the adaptation easier for the human user who should also be able to exploit PCTE specific facilities such as the control of Activities and the establishing of Working Schemas.

The migration of project/user data from existing File System structures into the PCTE's Object Management System (OMS) base can easily be achieved by copying the hierarchical File System structure into the OMS. Such structure could later on gradually evolve to exploit the augmented power of the data model, for instance by establishing additional relationships representing non-hierarchical dependences among objects; the original hierarchical structure (as well as the extended one) can still be accessed by tools developed to run on UNIX.

The explicit goal of the PCTE project is to support directly the immediate reusability of tools developed to run on System V (and thus of all UNIX tools which do not rely on some facility specific to a given, non System V version of UNIX). Thus the existing tools, as for data items, can be installed in the OMS base and can be executed in the PCTE framework. Given the number of tools one can expect to find on a given UNIX installation, the only suitable avenue (from the user point of view) is an installation procedure which does not require to modify the tools. Actually, the best solution is the installation of tools by just copying the tool executable representation into the OMS base.

In general, UNIX based tool sets will not exhibit the properties one expects for proper Software Engineering Environments, especially in terms of integration, of management of complex information bases and of friendliness of the tool <=> user interfaces: new tools, designed and developed to take advantage of the advanced PCTE facilities will gradually become available to be integrated into consistent SEEs. However, the ability to reuse existing tools will play a paramount role in the initial, critical phases in which transition to PCTE based SEEs has to take place.

5.2. UNIX compatibilty

Compatibility with UNIX in the context of the emerging standardisation initiatives (System V Interface Definition and X/OPEN/GROUP) has been a clear mandate for the design of the PCTE basic mechanisms: compatibility with UNIX is achieved by an emulation of the UNIX System Call functional level. Two complementary strategies are adopted:

- Some basic PCTE facilities have a one-to-one correspondence with UNIX primitives: these keep the same interface specification (the "C" language function definition, type and number of formal parameters) as the corresponding UNIX primitive. However the semantics of the basic PCTE operations are in general extended vis-a-vis UNIX.

- UNIX primitives which have not a direct correspondence with some PCTE basic function are still included to serve the purpose of compatibility. However, the same functionalities may also be achieved by some, more general basic mechanism function.

- UNIX concepts such as file system, device, process, can also be interpreted in the context of PCTE, though the corresponding PCTE concepts are more general.

An important effort has been done so far, and it will very likely have to be continued, to insure the compatibility of PCTE functional specifications with the evolutions of the above mentioned standard definitions of UNIX interfaces.

5.3. Which level of compatibility?

Compatibility between PCTE basic mechanisms and UNIX system call interfaces allows the importation into PCTE of tools developed to run on UNIX; however, compatibility can be achieved at various levels according to the program representation which is addressed: source code representation, object format representation, executable format representations:

1) Source level compatibility: this level is addressed in the PCTE Functional Specification report. It allows the transport of tools for which the source code is available by recompilation (and possible source code modification) of

each tool to be ported to PCTE. The source level compatibility is the easiest to achieve (from the PCTE implementors point of view); however, it implies the most difficult procedure to transport tools to PCTE: first, in general, source code is not available for tools which are commercial products; second, the compilation of several tools (there are hundreds of tools on a typical UNIX installation) may in itself be a non trivial task which involves making available the appropriate "makefiles", the included files and the (sources of the) libraries which are involved.

2) Object level representation (the ".o" one); this level, as one could expect, can be used as a base for porting tools across compatible hardwares. The object code level compatibility implies a simplified procedure for installing UNIX tools, however the availability of object code of tools can still be a real problem.

   The object code compatibility is nevertheless a candidate strategy for PCTE implementations, which presents however some additional drawbacks, as discussed later on in this paper.

3) Executable format compatibility; this plays a very important role because of its very attractive characteristic of allowing the easiest procedure for moving programs between different environments and in particular the importing of existing tools into PCTE: tools need just to be copied in the OMS space and are ready to be run via PCTE execution primitives. On the other hand, executable format compatibility implies the ability of capturing all UNIX system calls at SVC level (trap) to redirect them to PCTE. Thus the UNIX kernel would have to be relinked together with the PCTE basic mechanisms and the SVC table (as found in sysent.c, for unix gurus) to be extended to include the PCTE functions.

5.4. Implementation approaches

Two approaches are envisaged for the UNIX based implementations of the PCTE interfaces: these will exploit the object and executable compatibility levels as specific solutions for two different classes of host environments:

a) Add-on to the UNIX kernel: the standard System V (AT&T) implementation is the reference environment on which PCTE can be ported at virtually no cost with the best results. In this approach, the PCTE kernel becomes a true extension to the UNIX one. Thus, tools which are already available on a given System V installation are compatible at the executable format level and can be directly imported into PCTE. To give an example of the importance of this last aspect, there are some 400 tools already available under System V on the Bull SPS7; few super-gurus (if any at all) know all these tools at the level that could otherwise be needed to transport them to PCTE.

   Furthermore, such an approach can take advantage of the System V structure to gain on efficiency, especially in the connection with a critical area such as the OMS. However, the boundary between PCTE and System V software is clearly defined (and respected) so that there is no direct dependence between PCTE and System V internals.

b) "Black Box" implementation: this term is used here to designate an implementation of PCTE basic mechanisms which lies outside the O.S. kernel and is organised as a collection of user level processes. The principal problems which are associated with this approach are:

   - existing UNIX tools are to be relinked with new system call libraries in order to be imported into PCTE.

- performance: all basic operations, including evaluations of path names would imply passing around several messages; a less efficient mapping of OMS "volumes" on top of UNIX File Systems is implied.

- The two levels, the PCTE and the UNIX interfaces are both accessible by tools; thus PCTE users and the OMS data space need to be protected against native O.S users for consistency and security aspects. These issues present various fascinating implementation problems which are not yet entirely solved.

5.5. The PCTE prototype

An additional aspect of the project which is mentioned to complete the picture of the PCTE development strategy is the prototype implementation of PCTE interfaces which has been carried out in parallel with the above mentioned UNIX based phase.

The purposes of an early, prototype implementation of the PCTE interfaces are:

- it can serve inside the PCTE project as the vehicle for the verification of the completeness and the consistency of the PCTE interfaces;

- it can promote the usage of PCTE by providing to time critical ESPRIT projects a sound basis for the comprehension and experimentation of PCTE mechanisms;

- it can be a basis for the "Black Box" implementation of PCTE interfaces.

The goals of the prototype are thus to make available an <u>emulation</u> of the PCTE interfaces quickly and on a wide range of host environments. The achievement of such an objective has been possible because of two important design decisions:

- implementation problems are simplified by not having to address strict reliability and efficiency issues and concentrating the implementation efforts on a (significant) subset of PCTE facilities;

- the prototype implementation is based on an adaptation of the results of the Portable Ada Programming System (PAPS) project, which has been developed in the context of the CEC Multi Annual Programme.

The major emphasis has been on portability of the prototype across different versions of UNIX. The prototype has been so far successfully ported on:

- System V
- Berkeley 4.1 and 4.2
- Interactive System III
- MOSX
- MUNIX 1.4 and 1.5
- ULTRIX

6. Conclusions

We discussed the principal aspects of PCTE in connection with its immediate and long term strategies: <u>compatibility</u> with UNIX and adequateness of facilities to the needs of modern SEEs are the key factors in the short term; eventually, <u>portability</u> of PCTE basic mechanisms will become the principal aspect as the mean to preserve investments against obsolescence as new hardware and new operating

Thus, the short term approach is centered around tool portability and UNIX System V; the long term strategy has been concentrated on system portability adopting Ada and an architecture which stresses the independence of the PCTE implementation from specific operating systems.

However, the initiative will succeed only if the transition to PCTE based SEEs will be demonstrated to be a suitable avenue. This is indeed the challenge of the initial, UNIX based phase of the PCTE project.

Overview of
PCTE:
A Basis for a Portable Common Tool Environment

PCTE project team+

This paper is an overview of PCTE: it outlines an user view of the basic features of PCTE. The focus of the discussion is on the three principal aspects of PCTE: the Object Management System, the User Interface and the Distribution mechanism.

1. The PCTE project

The project "Basis for a Portable Common Tool Environment (PCTE)" is carried out by a consortium led by Bull (France) and including GEC and ICL (United Kingdom), Nixdorf and Siemens (Federal Republic of Germany) and Olivetti (Italy). The purpose of the project is to design and implement a software system to serve as basis for the development of complete, modern Software Engineering Environments.

The project started at the end of of 1983 and is to run for a period of four years. Within the Consortium, Bull, ICL and Siemens jointly develop the UNIX* based PCTE version; Olivetti is responsible for the early implementation of a PCTE prototype and for a longer term Ada** version of PCTE; GEC and Nixdorf develop two sample tools: the Knowledge Based Programmer Assistant and the Configuration Management System. These tools will exercise and demonstrate the validity of the PCTE design.

The project milestones and deliverables are defined so that intermediate results from the project can become visible and available to the ESPRIT community in time to be the basis for the integration and the dissemination of the results of other R&D projects. The first result of the project has been the production of the PCTE Functional Specification Report. The F.S. report gives the detailed definition of the PCTE functionalities in a form which can directly be used in the design of tools and programs which will eventually be integrated into the PCTE hosting framework.

PCTE will be developed in close cooperation with other ESPRIT projects, in particular with GRASPIN (graphical specification and formal implementation of non-sequential Systems) and SPMMS (Software Production and Maintenance Management System) and the ROSE infrastructure project.

+ contact person: Mr. J.P. Bourguignon
PCTE programme manager
Bull - DRTG/GL - 58P23
68, Route de Versailles
78430 Louveciennes
FRANCE

* UNIX is a trademark of AT&T Bell Laboratories

** Ada is a trademark of the Ada Joint Program Office

2. Overview of the basic PCTE facilities

PCTE is an hosting structure designed to be the basis for the construction of modern Software Engineering Environments (SEE). Each PCTE based SEE is regarded as an integrated collections of tools and services specific to a particular project life cicle model and/or application domain.

In the model architecture implied by the PCTE approach there is a clear separation between the tools and the underlying structure that hosts them. Indeed, some seemingly central aspects of the environment, such as a command processor, are relegated to the tool level.

The public, common tool interface to the PCTE servicies is defined by a set of program-callable primitives which support the execution of programs in terms of a virtual, machine independent level of comprehensive facilities. The following sections presents the principal aspects of PCTE namely:

- The Basic Mechanisms: these correspond to the functionalities required to manipulate the various entities that can exist in a development context. These entities are essentially programs that can be executed (typically tools, or programs under test), and various objects manipulated by the programs (data objects, such as various representations of the programs being developed, the documentation, input and output data; and also physical objects such as terminals and device drivers).
- The User Interface: above the basic mechanisms, which deal with the internals of tools, PCTE provides a number of facilities to assist in the construction of various aspects of the tools that will be directly visible to the end user. These aspects concern the forms of dialogue and of interactions (exchange of information) between users and tools, and the graphic facilities.
- Distribution: although not a direct concern to the user, the implementation of the environment on a network of workstations requires the definition of mechanisms and protocols to support the transparency of the distribution.

3. Basic Mechanisms

The PCTE Basic Mechanisms are subdivided into five categories: Execution, Communication, Inter Process Communication, Object Management System and Activities.

3.1. Execution mechanisms

The execution primitives deal with the notion of a program in execution. They define how an execution can be started or terminated, how it can be controlled, how parameters can be passed to the program, and more generally define the relations between a running program and the environment within which it executes. Facilities are provided for running a program (process), for defining and controlling the interactions between a program and its surrounding context.

3.2. Communication mechanisms

The communication primitives deal with the way a program can access the file type unstructured data (contents of objects) which are kept in the Object Management System database. They correspond to conventional input-output facilities and are closely modelled on those of UNIX.

3.3. Inter Process Communication

Special mechanisms are provided to allow different processes to exchange information. Although these could be viewed as part of the communication facilities, or of the execution facilities, they are treated specially because they play an important role in the implementation of the rest of the system.

In addition to traditional UNIX pipes and signals, PCTE provides a message-passing facility, and the possibility to share memory segments between users. These mechanisms are upward compatible with the ones found in UNIX System V, although they offer some additional, more powerful functionalities.

3.4. The Object Management System

A key aspect of an environment, and one that has a major impact on the complexity of the tool-writing process, is the set of functions that are provided to manipulate the various objects in the system.

The various "agents" in the environment (users and programs) "operate" on a number of entities that are known to the system and can be designated in it, and that are globally referred to as "objects". These may be files in the traditional sense, peripherals, pipes, or the description of the static contents of a program, but also objects representing information items such as project milestones, tasks, project management and progression records.

In a medium sized project, a huge number of objects are created which may have complex relationships. Among the numerous examples, one could mention:

- the documentation and the source code of a program (the latter may itself contain several modules);

- the history and the derivation trails for a given version of a given object (representing, a program, a program fragment, a document or other items to which Configuration Management can be applied).

- the test set for exercising a particular version of a module with the set of sample results that are supposed to be produced by these tests.

A uniform treatment of the various classes of objects, and powerful mechanisms to store and designate these objects, are two important requirements on a software engineering environment. The natural solution is a system that allows the user to associate a number of attributes, whose values represent specific properties, to objects, and to represent the various relationships which can exist between objects.

The basic OMS model is derived from the Entitity Relationship data model (ER) and defines Objects and Relationships as being the basic items of the environment information base.

Objects are entities (in the ER sense) which can be designated, and can optionally be characterised by

- a "contents" i.e. a repository of unstructured data implementin the traditional UNIX "file" concept;

- a set of attributes that are primitive values which can be named individually;

- a set of relationships in which the object participates.

Relationships allow the representation of logical associations/dependences between objects as well as structured information. In particular, one might need to introduce new compound objects, composed of several objects (a notion that supercedes the traditional directory), or to establish explicitly a reference from one object to another. Relationships also may have attributes, which can be used to describe specific properties associated with the relationship, and which allow the designation of a specific relationship (among the possibly many in which a given object may participate) by the values of its (key) attributes.

Designation of relationships is the basis for the designation of objects: the principal means for accessing objects in most OMS operations is to navigate the OMS object space by "traversing" a sequence of relationships designated by a string value type pathname. The syntax induced by the OMS on such strings is (of course) compatible with the syntax of UNIX pathnames.

Objects and relationships have a type which defines their basic properties. Object, attribute and relationship type definitions are contained in special objects of the predefined type Schema Definition Set (SDS). SDSs can be specific to individual users and/or tools or be common to a community (of users or tools).

At any time, a process operates with a set of visible definitions, that constitute its working schema. The working schema is established for a process as the well-formed composition of a set of SDSs and can be dynamically re-established so as to change the working context of a process. A working schema corresponds to a description of certain constraints on the properties of a collection of objects and of relationships which are operated by a given process. A working schema can thus reflect the specificities of a given tool when applied to the objects and relationships specific to a given user.

Thus, different users and tools may operate with different working schemas, though accessing the same object, resulting in the facility to particularise the way in which an object is "seen".

These mechanisms resemble in many ways to a full-fledged DBMS, with some significant differences:

- The goal of the system is not to make complex computations with the values of the attributes that are associated with an object: the major part of the computation will be performed on the object contents, whose details are administered by the tools.

- The user must have the possibility to use the system for his own needs, and to modify its working context by the definition of new object and relationship types, without having to go to a higher authority.

3.5. Activities

The lack of data access synchronisation and recovery mechanisms is one of the recognised deficiencies of UNIX like environments. It is overcome in PCTE by adapting the well known notion of transaction to the context of Software Engineering Environments. In a general sense, a transaction can be regarded as a workframe in which individual operations take place. A transaction can be defined to have certain properties, namely

- it is atomic, the effect on data of operations performed on behalf of a transaction is either applied as a whole or is not applied.

- it is serialisable, the effects of executing several transactions to operate on the same data domain at the "same" time are expected to be the same as if the same transactions were executed in mutual exclusion.

In PCTE, the concept of transaction is generalised to the concept of "Activity".

One important and distinguising aspect which is dealt with by PCTE Activities is the concept of "granularity" of tools, which means regarding each tool (either program, script or program fragment) as a modular component, performing a well defined (hopefully atomic) function. More powerful tools can be assembled out of simpler ones; the new tool can become itself a new grain participating in the construction of other tools.

Each tool can be regarded as a primitive; functional layers can in this way be built out of tool sets. The failure of one of the tools may or may not be interpreted as a complete failure depending on the context in which it was invoked. The natural solution is to allow for nesting of Activities, in a way similar to the dynamic first-in-last-out management of the local context of blocks and procedures in modern programming languages. Thus, PCTE Activities can be nested; the basic PCTE facilities support a complete and consistent model featuring implicit as well as explicit recovery and synchronisation mechanisms (locks) which fully support the atomicity and serialisability requirements. Furthermore, Activity types and object level operations are supported which allow the tool writer to tune the mechanisms to the desired level of data consistency and concurrency control, ranging from the UNIX like "unprotected" behaviour to proper, fully protected atomic and serialisable "transactions".

Activities are meant to support in a systematic and standard way the construction of robust tools.

4. The User Interface

4.1. Introduction

The design of a User Interface is becoming more and more important along with the development of hardware being able to drive graphic output. The availability of workstation computers with a high resolution raster screen and a pointing device (e.g. a mouse) makes it possible to develop interfaces that at least have the feature to produce user friendly output instead of plain text. This paper talks about the basics that a User Interface should provide in general, using the resources of the computer in a sound way. It is not concerned with a discussion of the conceptual power that a User Interface may employ [2][4]. Many principles, of course, remain the same.

Since PCTE concepts are based on modern operating systems, several aplications may run in parallel. All of them may require interaction with the user. The User Interface provides the user with a communication port to the different applications in the user's working configuration, interprets user input, and channels it to the corresponding applications. This is supported by dividing the visible screen into several rectangular areas called windows, whereby each window may serve as an output media for an application, and by providing powerful communication mechanisms for application interaction.

4.2. An object oriented User Interface

The User Interface will work in an object oriented way. The objects formed by the User Interface are windows, icons, or the elements of different datatypes such as characters, strings, graphical elements, etc. The basic interaction mode is represented by first selecting an object and second performing an operation on the selected object. For instance, the user may select any object currently visible in any window on the screen, specify the destination (which does not have to be the same window) and issue the generic copy-command. The User Interface generates information about the type of the selection and hence it is able to determine which specific actions are required to perform the generic copy.

This principle is extended from the idea of object oriented programming [1]. However, the User Interface is based neither on such a programming language nor an object oriented operating system. It is the method of user interaction which is object oriented. At this point it should be stated that the objects formed by the User Interface are not managed by the PCTE Object Management System. In fact, objects within the User Interface are dynamic data structures, which are managed locally.

The User Interface provides the standard functionality of a window management system. Windows can be thought of as pieces of paper arranged on a desk. Windows can be moved around on a desk. Windows may overlap or overlay on the screen to utilize the screen optimally by hiding non relevant information. Every application can be accessed by the user via its associated windows. It is the responsibility of the User

input to an application can be achieved from the user point of view either by input from the keyboard, pointing device, or by copying any selected item.

It is also possible to replace a window by a small placeholder, the icon, which indicates to the user that a running application has no window for its physical representation. In this case the output of the application is not visualized. Icons permit the saving of screen space for running or waiting applications and allow easy re-activation mechanisms for communication to the user. They may also be used to provide an iconic interface, as for instance in the Star system [3].

Applications run independently of one another and are able to communicate with each other at a low level by passing messages. PCTE is based on the low level message passing system as it is described in PCTE Basic Mechanisms. Based on the active role, the entities of the User Interface are classified as:

- User Agent

- Applications.

Figure 1 illustrates the basic structure of the User Interface model. The User Agent is in charge of translating the intentions of the user to the system. Its functionalities cover the display and the management of windows, the control of input from various input devices and the management of commands and menus. The most important work however is the synchronisation within the User Interface.

Applications represent tools to the User. Existing tools will use an OS-Application, emulating the host operating system in a window working in a standard terminal mode.

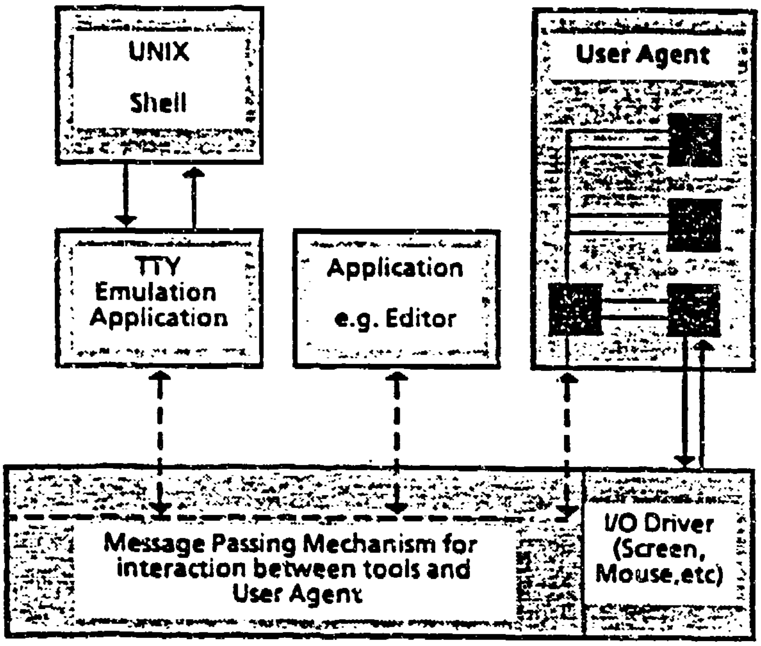


Figure 1 Basic Structure of the User Interface

These tools may then be used within the User Interface without any changes. However, they will in general not be able to make use of the advanced functionalities offered in the User Interface.

4.3. UNIX versus PCTE

The first approach to implement PCTE is to take UNIX as a basis. It is obvious that most existing UNIX tools for alphanumeric displays do not know about windowing systems, object oriented handling, etc. It would be a major task to convert them all. Instead a UNIX TTY emulation must be available. This emulation provides one window for character input and output, having the behaviour of a standard alphanumeric terminal. In addition, this TTY emulation allows the user to select text displayed within the window. Thus it is possible to copy text from the TTY window to any other tool. In the same manner, the user may perform free editing of the current command input line.

The exploitation of the User Interface will be done by new tools to be developed especially for this environment. It should be noted that at present many vendors already offer windowing facilities and enhanced graphic output for UNIX. Most of them have developed highly sophisticated tools, such as display oriented text editors, document preparation systems, font editors, interactive program environments etc. However most of them are written specifically for the given workstation. It is obvious that these tools are not portable at all and hence cannot run in the PCTE environment.

4.4. The User Interface from the user's point of view

We have to separate the PCTE users into two groups: one consisting of users working with tools (e.g. editing a document, printing mail, etc.) and the other consisting of tool writers. Their task is the development and implementation of new tools. They work with the primitives provided for programmers as they are described in the Functional specification of PCTE. This paper is concerned with the first group.

When initializing the User Interface of PCTE, the screen will be presented as an empty grey area. If the User is not already logged into the system, he will be requested to do so first. This task is controlled by a new login tool, running with a special login window. Any user may request a command script to run after identification.

This method allows the generation of a specific tool environment to be initialized for each login identification. A discussion on command scripts follows in the chapter of "command language".

The interaction between the User Interface and the user may be classified as follows:

- Tool Management

- Window Management

- Basic Editing

4.4.1. Tool Management

Once a UNIX shell is running, the user will find himself in the standard UNIX environment. No special support is given as long as he is not using any of the tools specifically written for PCTE.

Any new tool normally starts up by creating it's own window (or set of windows) providing a new working environment. The behaviour, of course, is determined by the characteristics of the tool.

The User Agent provides facilities to allow the execution of tools without having to run a shell. The profile for any user may define tools that may be started directly. Selecting one of these tools by name from a system defined menu will result in starting the tool. Further control is done by PCTE primitives or by a special PCTE process manager.

4.5. Window Management

Windows are User Interface objects. They may react to any generic command the user may have initiated. The communication between the tools and the User Agent is achieved by sending messages. Each command generated results in a message sent to the appropriate tool, which is in charge of the object.

Figure 2 shows an example of the move command. The advantage of this concept is that any tool may control any operation the user initiates and check for constraints (e.g. don't move a window so that it hides relevant parts on the screen.

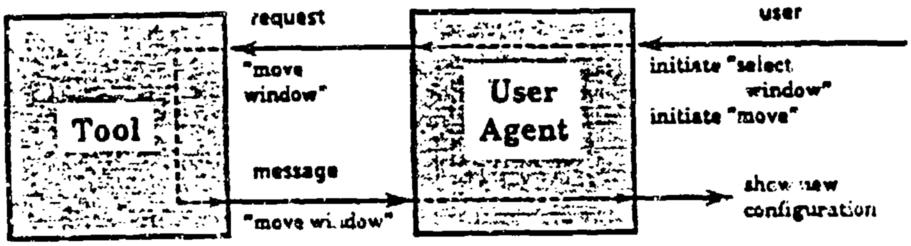


Figure 2 An example of the move command

Examples of operations on which windows may react are:

- Create Window

  A tool may create a window on an already existing data structure. Depending on the characteristics required, the size and position may be requested interactively or may be predefined. It is important to note, that a user should not be able to create a window explicitly to start a tool. The end user should only start the tool. It is the responsibility of the tool to determine if a window is necessary. In this manner the tool is able to place a window if necessary at a predefined position or ask the user to specify the location or the size. In contrast, a tool may start as being represented by an icon only.

- Iconize Window

  An open window will be replaced by a specified icon at a position which is defined by the user on the screen. Icons may be moved and hidden on the screen like windows.

- Expanding an Iconized Window

  An icon is expanded to a window with the same characteristics as before iconizing (however, a change of position and size may be employed). The contents of the window are updated according to the viewport, reflecting any changes that may have occurred while the window was iconized.

- Additional Operations

  In addition, all standard window management operations, such as delete, move, bury, etc are supported.

Even when a window is iconized, it may still be sensitive to messages. For instance, a "file directory icon" may be responding to an insert command with the argument being another "file icon". Expanding such a "file icon" may_ invoke the text document preparation system on this file. This enables the creation of a User Interface environment in a manner which is commonly used in office automation systems [3].

4.5.1. Basic editing

Basic Editing is provided to allow simple and easy modification of objects visible on the screen. Strictly speaking, window management is part Basic editing. Moving a window on the screen is essentially not different from moving a character from one position to another. This section, however describes the facilities provided for the management of objects independent of their data type:

- Selection of an object

  Simple selection is defined for each type by default. It may be replaced by more powerful selection functionalities in special tools as needed. The selected objects will be highlighted to visualize the selection to the user. There are three kinds of interaction:

  - start selection

    A new selection will be defined. If an object is already selected, it will first be deselected. The position of the cursor determines the new object to be selected. There is only one selected object in the syten. At the time the object is identified, it must also be visible on the screen.

  - multi-click selection

    Pressing the start selection button within an already selected item will increase the selection level. For instance, a syntax oriented editor may identify each level with a production in the underlying grammar. On the other hand, multi-click selection is not defined for ordinary windows.

  - Extend selection

    The current selection will be extended to the position of the cursor at the current selection level. For instance, if a line is the current selection and the user presses the extend selection button on the next line, then these two lines are selected. For graphics etc, the extend selection may spawn a rectangle to include all objects inside the rectangle for selection.

- Copy Selection

  The contents of the current selection will be inserted according to the definition at the insertion position. Type conversion of objects (e.g. copy from a simple text frame to a complex text) is applied if it is appropriate. The copy of the objects window and icon is not supported.

- Delete Selection

  The contents of the selection will be deleted. the selection itself will be deleted, too, since it is no longer meaninful.

- Move Selection

  This functionality is a combination of Copy Selection and Delete Selection. However, for certain objects (such as window, etc) the operation will be performed directly.

4.5.2. The current window

Even though many tools may run in parallel within PCTE, there is only one keyboard and one locator input device for all of them. The sharing of these input devices is mandatory. The basic interaction for the user is as follows: he will have to select the object window (and hence the corresponding tool) and issue the "keyboard pressed" command at the time a press on the keyboard is performed. Even though this technique seems to fit into the model, it is somewhat unnatural and makes certain kinds of interaction slow. For instance, to copy a string from one place to another, the user will have to perform the selection of the string first. Since there may be only one selected object in the User Interface, any possibly selected window will be deselected first. After issuing the copy command, a destination position has to be specified. When the copy is complete, there are two choices: either the selected string remains selected or it will be deselected. In any case, to continue with normal data input, a window has to be reselected. Any task will be slowed down heavily by the repeated reselection of windows after issuing a generic command.

The solution is to define a current window independent of the selected window. A window will remain current unless it is destroyed or another window is defined to be current. Any character input will be read by the current window, independent of the selected object. This enables the user to type into one window and use generic commands for any kind of objects at the same time.

In this respect, the tool having the current window is said to be the current tool.

4.6. Command Language

It has been demonstrated that the User Interface may work directly by user control of certain peripheral devices. For instance, pressing a certain button on the mouse may select an object on the screen. Any following command will then be sent to the tool managing the command.

This chapter talks about how commands may be entered and how tools will read them.

The User Interface defines several input classes. The most important are:

- Keyboard:

  Conventional character data input may be read within this class.

- locator:

  This class provides a mechanism to read positions on the screen, either in screen coordinates or (which is usefull for tools) in window coordinates. This input is typically provided by a mouse.

- message:

  This class is used for communication between a tool and the User Agent. Elements in this class are generated by the User Agent. The task of generating message usually depends on input from the classes keyboard and locator.

  The User Agent has to determine if the input from keyboard and locator class is to be passed through to a tool directly or if it has to be interpreted to form a message. Figure 3 shows the flow of data.

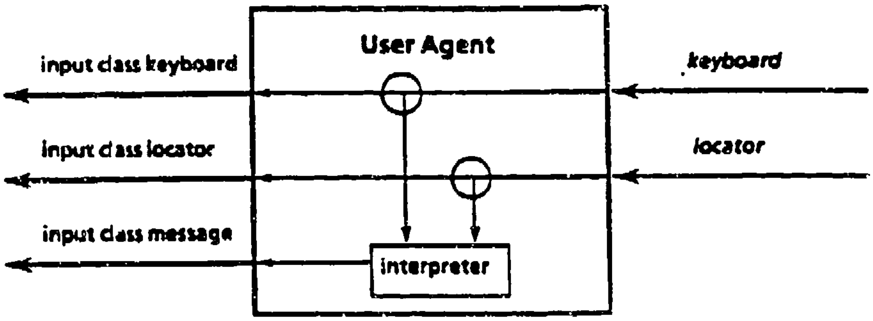


Figure 3 Input data flow

The interpreter within the User Agent determines the behaviour of the User Interface. This behaviour of course is fixed and cannot be altered. In addition, it is not possible to enter any command in text form. To get around this problem a command language interpreter may be installed as an application. This command language application may read from the User Agent in the keyboard class, parse the input and generate output in any other class to be sent to the User Agent for distribution to another tool. Figure 4 illustrates the set up.

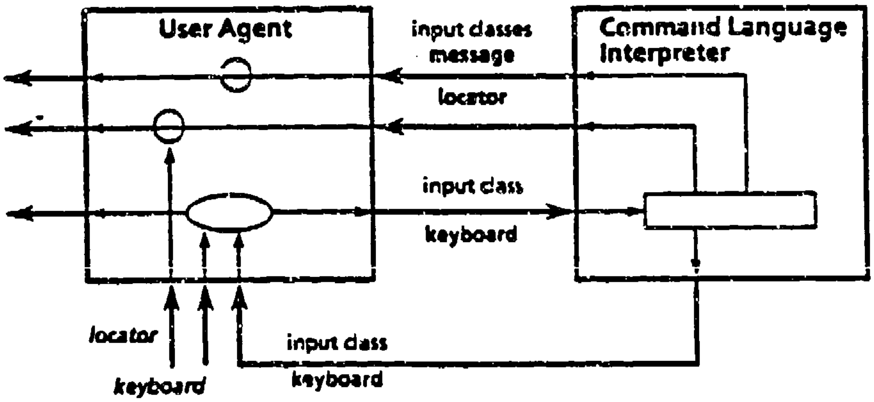


Figure 4 Input data flow

This technique allows the implementation of user tailored command language tools and enables many features:

- produce a history list of commands in a user readable format,
- provide for execution of command scripts,

- allow easy control of one tool by another tool,

- provide an interactive, interpretative environment for User Interface issues.

5. Distribution in PCTE

The PCTE preferred architecture is a Local Area Network of powerful single-user workstations and associated resources. In this environment project teams and users share software, data, and peripherals. This whole architecture is seen by the user as a single machine and yet each machine can act as autonomous unit.

All workstations in the environment have equal rights. Successful operation of the Distribution mechanism does not depend upon any hierarchy, fixed or configurable, of the workstations or of the underlying LAN. Nor is the ability of a process to make a request of a remote kernel altered or diminished by changes in the LAN topology.

The Distribution mechanism provides:

- Transparent distribution of the functions of the OMS, process execution, process structure and inter process communications.

- Network Administration ie. control management of the communications layer, namely the OSI Transport.

- Distribution Management, ie. control and configuration of the distribution mechanism itself.

5.1. Process Distribution

The transparency of distribution imposes a generalised access to various processes running in the system. A process is characterised in PCTE by its executable code, its data and a collection of information constituting its "execution context" and is uniquely identified by its system-wide process_id.

No process may migrate from the workstation on which it was created, but this is no limitation on its ability to access remote resources. The Distribution mechanism allows a process to create child processes on other workstations, either deliberately (by explicitly quoting the desired station_id) or implicitly, according to the characteristics of the child itself, such as its specific hardware requirements.

Similarly, when accessing OMS objects, the process need not be aware of the physical locations of these objects. The Distribution mechanism shields the process from being dependent upon topological considerations.

5.2. Objects Distribution

The OMS database is partitioned into volumes, which correspond to logical disks (equivalent to the notion of File System in UNIX). Each volume is represented by a specially typed OMS object. In principle (hardware permitting), each volume can be dynamically allocated ("mounted") to a particular workstation. On mounting, a volume is immediately visible to the whole of the system as in Locus [8]. The distributed database is managed by the static and dynamic management of volumes: statically by their creation and deletion in the OMS, dynamically by the mounting and unmounting of a volume.

Each workstation is considered a potential server to the other workstations for the portion of the object base which is contained on volumes "mounted" at that workstation. Relationships can exist between objects contained on different volumes. Access to objects on a mounted volume is made by navigating through a sequence of links, originating from one of the process's reference nodes. It is the responsibility of the distribution software in association with OMS to provide access to every object available to each particular user, both in locating where it resides and in providing the remote access methods. Actions performed on remote objects should appear to the user in every respect to have the same effect as if applied to a local object.

There is no need in PCTE for the concept of a system wide "super-root" as in the Newcastle Connection [9], or of "special network files" acting as local dummies for remote roots as in Cocanet [10].

5.3. Distribution of Activities

The concurrency and integrity control facility of PCTE (Activities) also operates in a distributed context. System wide identification of activity contexts, management of volume level logging as well as the distributed termination of activities (two phase commit/roll-back protocol) are supported by the distribution mechanism.

5.4. Inter Process Communication

As a process structure becomes distributed so may its means of Inter Process Communication (IPC). All forms of IPC supported by PCTE basic mechanisms may be distributed that is named and unnamed pipes, named and unnamed message queues.

Named pipes and named message queues are special OMS objects and consequently are managed as such. Unnamed pipes and message queues become distributed by the creation of a distributed process structure. A user need not be aware, by use, that IPC is distributed.

5.5. Network management

Each workstation is represented by an object in the database, and is also identified in a globally unique way by the identification number (host_id). This number is assigned when the work station is initially created. All PCTE primitives that circumvent the transparency of the distributed system make explicit use of the host_id.

The Distribution mechanism has to manage the static topology of workstations as well as the connection and disconnection of workstations from the system.

The Distribution mechanism itself requires configuration and management, in order that it may act as efficiently as possible in particular installations, and that it may hide the installation's physical topology and provide an invisible service. This information has to be maintained in a replicated form on all workstations as each station is to be permitted to function autonomously.

5.6. Working in the distributed PCTE.

This section describes what a user sees of the environment on connecting to a PCTE.

A user is known by his name, password, identity and group number, his reference objects (eg. initial home working directory) and his initial Working Schema. The password file contains the details of each recognised user.

The Newcastle Connection maps between distinct password files located on different machines, whereas Locus maintains in an automatic way "logical file groups" that are to be found replicated throughout the environment as "physical file groups". Our design follows more closely the approach of Locus: unlike the Newcastle Connection, in PCTE a user has a single entry in a password file which is common to all workstations. This approach makes the stations shareable, and consequently any known user may work on any of the connected workstations.

After start up and connecting a workstation to a PCTE to which he belongs a workstation user can access, in a totally uniform way, all the resources of the entire environment (software, hardware and objects found on mounted volumes) to which he is entitled as if they were on his own workstation. This fairly obvious statement does have important implications both in the philosophy of what PCTE exactly is and in the implementation strategy chosen: a PCTE is not a community of separate communicating computers as in the Newcastle Connection. The binding within PCTE is considerably tighter. Member of the PCTE community are unaware of the communication between them and share such entities as password and group files, as they do with all system administration information. Information, with few exceptions, is distributed throughout the environment.

There is no hierarchy of workstations implied in the structure. A workstation need only have access to sufficient system administration information to enable it to work (in isolation or otherwise).

5.7. Architecture of the Distribution mechanism.

The PCTE kernel is built on top of the UNIX kernel. The implementation of the distributed environment does not depend on a distributed version of the kernel, but on the presence of facilities for the construction of a system of communicating kernels. The Distribution mechanism is part of the kernel, and as such exists on each workstation; in this sense the system corresponds to the Newcastle Connection [9].

Processes use remote resources by means of a master-slave process pair, as in Locus rather than the Newcastle Connection. Internal communication at the level of kernels (IKC) is handled by a formal, stylised Remote Procedure Call (RPC) mechanism. This RPC mechanism relies on a service providing both connected and connectionless transport facilities (supplied from the Esprit ROSE [5] project).

A PCTE system is essentially restricted to a LAN. The LAN development has been simplified by making certain assumptions:

- terminals are only attached to PCTE via a PCTE workstation. An exception to this may be target computers attached to the LAN for embedded system development.

- every workstation must provide an implementation of ISO OSI layers 1 and 2 for CSMA-CD.

- the ROSE (Research into Open Systems for Europe) project will provide an implementation of OSI Transport Service and Protocol(class 4) conforming to ISO DIS8072 and DIS8073 and a Connectionless Transport Layer conforming to emerging ISO standards.

The Distribution mechanism is being designed and implemented in line with present and emerging ISO Standards, and will use ROSE Transport for all LAN access.

5.8. PCTE Gateway

So far we have described only the mechanism that binds all the separate workstations into a single unit, a PCTE, but not described how a PCTE may communicate to another system or PCTE.

A PCTE gateway is a software mechanism that can be configured to communicate at a particular logical level, through a known communication device, to a known recipient. In this connection, gateway functions make available to the user the Session and Presentation layer of the ISO model, supporting, for instance:

- A session level communication via WAN to another PCTE.
- An application level file transfer via the LAN to a mainframe.
- A guest user having limited capability using a remote computer.

5.9. Summary of disitribution facilities

- PCTE is LAN based.
- A PCTE can act as if one single machine.
- The OMS database, Activities, IPC and process context are transparently distributed.
- Certain topologies are known and managed system wide, namely workstations and volumes.
- Internal PCTE communications are ISO OSI using, in particular, Esprit ROSE software; this ensures maximum portability of the PCTE implementation across a different machines.

6. References

[1] Goldberg, A.: SMALLTALK 80. Addison-Wesley (1984)

[2] Good M.: Etude and the Folklore of User Interface Design. SIGPLAN Not., Vol. 16, No. 6 (1981) 34-43

[3] Smith, D.C.; Irby, C.; Kimball R.; Verplank B.; Harslem E.; Designing the Star User Interface. BYTE, April (1982) 242-282

[4] Thimbleby, H.: User Interface Design: Generative User Engineering Principes. In: Fundamentals of Human - Computer Interaction. Academic Press (1984) 166-179.

[5] ROSE Technical Specification Version 3 March 85.

[6] PCTE Functional Specification report - third edition

[7] Connectionless Transport

[8] Locus. A Network Transparent, High Reliability Distributed System. University of California. December 1981.

[9] The Newcastle Connection. Software - Practice and Experience, vol 12. February 1983.

[10] A Local Network Based on the Unix Operating System. IEEE transactions of Software Engineering, Volume SE-8 No 2. March 1982.

Ada-Europe

Ada-WG

Author:

Members of the Environment WG

Ada-Europe Document No.
Ada/WG/6/51 Issue 2

Date:
8th January, 1986

Distribution:

WG

WG Chairperson:

Mr. John C D Nissen
GEC Software Ltd.
132-135 Long Acre
LONDON WC2E 9AH, UK
Tele: +44 1 240 7171
Telex 21403 GEC LA G
Fax: +44 1 240 9329
uucp: ukc ! hrc63 ! gecsw ! trevor

Title:

PCTE Conformance to the RAC

Summary:

An evaluation of the Portable Common Tool Environment (PCTE) against the DoD "Requirement and Design Criteria for the Common APSE Interface Set"

* Ada is a registered trademark of the U.S. Government Ada Joint Program Office

Ada-Europe Chairman:

J G Glynn (Ada-Europe)
Ferranti Computer Systems
Systems Limited
Ty Coch Way
CWMBRAN, Gwent NP44 7XX
UK

Tele: +44 6333 71111
Ext 761 240
Telex: 497636 FERCWM G

Ada-Europe Secretary:

J-P Rosen
Ecole Nationale
Superieur Telecommunications
46 Rue Barrault 75634
Paris
Cedex 13

Tele: +33 1 589 666
Ext 4646
Telex: 200914 F

Ada-Europe Vice Chairman:

M Boasson (Ada-Europe)
Hollandse Signaalapparaten B.V.
P O Box 42
7550 GD Hengelo
NETHERLANDS

Tele: +31 74 482129

Telex: 44310

CONTENTS

SUMMARY 1

1. INTRODUCTION 2
   1.1 Scope 2
   1.2 Terminology 2
   1.3 Assumptions about PCTE support in Ada 3
   1.4 Assumptions about potential CAIS 2 support by PCTE 3
   1.5 Completeness 3

2. GENERAL DESIGN OBJECTIVES 4
   2.1 Scope of the CAIS. 4
   2.2 Basic Services. 4
   2.3 Implementability. 4
   2.4 Modularity. 5
   2.5 Extensibility. 5
   2.6 Technology Compatibility. 5
   2.7 Uniformity. 5
   2.8 Security. 6

3. GENERAL SYNTAX AND SEMANTICS 7
   3.1 Syntax 7
   3.2 Semantics 8

4. ENTITY MANAGEMENT SUPPORT 10
   4.1 Entities, Relationships, and Attributes. 11
   4.2 Typing. 12
   4.3 Identification 13
   4.4 Operations 14
   4.5 Transaction 16
   4.6 History 16
   4.7 Robustness and Restoration 16

5. PROGRAM EXECUTION FACILITIES 18
   5.1 Activation of Program 19
   5.2 Termination 20
   5.3 Communication 20
   5.4 Synchronization 20
   5.5 Monitoring 21

6. INPUT/OUTPUT 22
   6.1 Virtual I/O Devices: Data Unit Transmission 22
   6.2 Virtual I/O Devices: Data Block Transmission 23
   6.3 Datapath Control 24
   6.4 Data Unit Transmission 25
   6.5 Data Block Transmission 27
   6.6 Data Entity Transfer 28
   6.7 General Input/Output 28

7. ACKNOWLEDGEMENTS 29

SUMMARY

PCTE very largely meets the requirements of the RAC. It is especially strong in the entity management area, where it already provides typing and transaction facilities. There are a great many individual requirements in the RAC: of these, only a very few are not addressed by PCTE, and there are a few more where PCTE only partially meets the requirements.

The two main areas where PCTE does not meet the RAC are:

a. The need for Ada specifications.
These are underway.

b. Security.
PCTE's security mechanisms were not designed with DoD standards like CSC-STD-001-83 Class B3 criteria in mind. A careful study of this area is needed.

There are a few other aspects, including:

a. PCTE does not support triggering (4.2E).

b. PCTE does not provide exact identities (4.3A), although it could be extended to do so.

c. PCTE could only support task waiting (5.4A) by a mechanism that might be unacceptably inefficient, and would require modification to an Ada RTS (5.5B).
There are areas where our study felt that the RAC needs further consideration. It is not obvious that any CAIS could meet both requirements 5.4A and 5.5B.

d. PCTE does not fully provide the identification methods required in 4.3C.

One should also note that PCTE, as well as largely meeting the RAC requirements, also presents a similarity of concept with CAIS 1 in most areas. Thus an evolution of PCTE could conceivably meet the CAIS 2 requirement of upwards compatibility of CAIS 1.

1. INTRODUCTION

1.1 Scope

This document compares the *DoD Requirements and Design Criteria for the Common Ada*[1] *Programming Support Environment (APSE) Interface Set (CAIS)*[2] (known as the *RAC*) to the *Portable Common Tool Environment (PCTE)* Functional Specifications[3]. The CAIS and the RAC are products of the U.S. DoD-sponsored KAPSE Interface Team (KIT), and its companion Industry-Academia team (KITIA). The PCTE is a product of the Commission of the European Communities (CEC) European Strategic Programme for Research and Development in Information Technology (ESPRIT) program.

The RAC is a requirements document for a CAIS. While one could define a number of different "CAISes" which "comply" with the RAC, the U.S. DoD intends to provide one specific CAIS as a standard for the DoD and its contractor community. There is an initial CAIS known as *proposed DoD-STD-CAIS*[4]; this specific CAIS is informally also known as *CAIS 1;* it partially complies with the RAC. There is a current contract, let by the U.S. Naval Ocean Systems Center, to produce a follow-on product, known informally as *CAIS 2*. CAIS 2 is planned both to comply with the RAC and to be an upward compatible derivative from CAIS 1. It is planned that CAIS 2 will eventually become a U.S. DoD Standard, and as such be imposed on U.S. Government agencies and their suppliers.

There are no plans to use the RAC as a generic template for conforming CAISes; the only current purpose and *raison d'être* for the RAC is to define the requirements for the specific CAIS which will eventually replace the proposed DoD-STD-CAIS (i.e., to define CAIS 2).

A comparison of the RAC and PCTE is likely to yield some useful results. Firstly, PCTE contains models for some features of the RAC which are not defined in CAIS 1. A comparison of PCTE and the RAC will clearly identify these models and thus contribute to the CAIS 2 design process. The influence of these models on CAIS 2 may facilitate a convergence of PCTE and CAIS.

Secondly, since a PCTE implementation already exists, a comparison of the RAC and PCTE yields information on the implementability of RAC concepts. This may influence further development of the RAC and help to attain the general design objective of implementability.

This document will examine the conformance of the PCTE to the RAC. This requires some assumptions about the appearance of the PCTE as a set of Ada interfaces because the PCTE is currently defined only in the C language.

In some places, where it is felt helpful, specific comparisons are made between the PCTE and the CAIS.

1.2 Terminology

Precise and consistent use of terms has been attempted throughout the document, by use of the nomenclature and style of the RAC.

[1] Ada is a trademark of the U.S. Government, Ada Joint Program Office.

[2] "DoD Requirements and Design Criteria for the Common Ada Programming Support Environment (APSE) Interface Set (CAIS)", prepared by the KIT and KITIA for the U.S. Government, Ada Joint Program Office, September 13, 1985. This document is in a comment cycle and is not an official Government document.

[3] "PCTE A Basis for a Portable Common Tool Environment," Functional Specifications, Third Edition, Bull et. al., 1985.

[4] Military Standard Common APSE Interface Set (CAIS), U.S. Government, Ada Joint Program Office, 31 January 1985, Proposed MIL-STD-CAIS. This document was renamed a proposed DoD standard to provide wider applicability, during 1985.

The following verbs and verb phrases used by the RAC are borrowed and used throughout the document to indicate where and to what degree individual constraints apply. These phrases are used in the Conformance Requirement column of the summary tables in the following sections.

shall indicates a requirement on the corresponding definition.

shall provide indicates a requirement to provide the prescribed capabilities.

shal! support indicates a requirement to provide prescribed capabilities or for an implementation to provide that capability constructed from other provided interfaces.

should indicates a desired goal but one for which there is no objective test.

1.3 Assumptions about PCTE support in Ada

There is a significant difference in style, parameterization, and error handling between C language system calls (e.g., as defined by UNIX and PCTE), and the way system calls are handled in Ada (e.g., in the style of the Ada Reference Manual[5] and in the style of CAIS 1). Simple transliteration of the PCTE C interfaces into Ada would not result in "acceptable" Ada style.

We assume that PCTE support in Ada would be accomplished by translation into Ada style, both in terms of identifiers (e.g., meaningful mnemonics without the 7-letter restriction of C), in terms of parameters (e.g., use of appropriate typing), and use of exception conditions (e.g., instead of the UNIX error return convention). We further assume use of Ada's ability to distinguisl overloaded subroutine calls to reduce the number of discrete PCTE functions.

1.4 Assumptions about potential CAIS 2 support by PCTE

One of the pertinent requirements on the CAIS 2 contractor is that the new product be (to the greatest degree feasible) upward compatible with CAIS 1. The CAIS 2 is likely to depart from CAIS 1 in its specification of the RAC-required features not in CAIS 1; for example, the RAC requires typed entity management support, but does not specify a model for that support.

In those circumstances where PCTE provides a model for RAC feature implementation which goes beyond CAIS 1 (e.g., typed entity management support), we have made the assumption that the model is acceptable, but of course we have no way of predicting whether this model will actually be the one adopted in CAIS 2.

1.5 Completeness

For completeness, we have included all the sections of the RAC, and have inserted our comments in an italic typeface following each section.

[5] "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A, U.S. Department of Defense, January 1983.

2. GENERAL DESIGN OBJECTIVES

Section	Conformance	
	requirement	RAC to PCTE
2.1 Scope of the CAIS	shall provide	Yes
2.2 Basic Services	should provide	Yes
2.3 Implementability	shall	Yes
2.4 Modularity	should	Maybe not (see text)
2.5 Extensibility	should	Yes (libr. level)
2.6 Technology Compatibility	shall	International
2.7 Uniformity	should	Yes
2.8 Security	shall provide	Probably not (see text)

2.1 Scope of the CAIS.

The RAC requires that the CAIS shall provide interfaces sufficient to support the use of APSEs for wide classes of projects throughout their lifecycles and to promote I&T among APSEs.

PCTE is intended to serve for projects which may span the duration of the entire ESPRIT programme (approximately 10 years), and its successors. It is also intended for supporting industrial software projects, during their entire life-cycle; one can observe that one of the two sample tools funded in the PCTE programme is a configuration management system. The first class of projects to be supported are derived from the ESPRIT programme itself, and its five main thrusts (advanced microelectronics, software technology, advanced information processing, office automation, and flexible manufacturing). Furthermore, PCTE goes beyond the needs of an Ada-only PSE (APSE), since it should support multiple languages, and multi-language projects.

I&T has been given substantial consideration. In particular, the schema mechanisms and the multi-level schema definition sets are intended to guarantee that an entire subproject can be moved to a different environment without requiring a change in the naming of the objects, the attributes or the links (the lack of adequate name space management is one of the major deficiencies of CAIS 1 in this respect).

2.2 Basic Services.

The RAC requires that the CAIS should provide simple-to-use mechanisms for achieving common, simple actions. Features which support needs of less frequently used tools should be given secondary consideration.

Ease of use is one of the specific objectives of the PCTE project team. (It is a matter of taste as to whether a particular set of mechanisms is simple or not, easy to use, or not.)

However, giving secondary considerations to "less frequently used tools" is a dangerous policy for which PCTE has a different approach: it has been frequently observed that the "less frequently" used tools of yesterday have turned out to be important and popular. (A example outside the traditional software engineering area is the spreadsheet processor.) The PCTE designers have attempted to rely on advanced research (e.g., in the application of knowledge-based techniques to the software development area) to get a feeling for which features may turn out to be important in the future, and how they can be combined with today's needs.

2.3 Implementability.

The RAC requires that the CAIS specification shall be machine independent and implementation independent. The CAIS shall be implementable on bare machines and on machines with any of a variety of operating systems. The CAIS shall contain only interfaces which provide facilities which have been demonstrated in existing commercial or military software systems. CAIS features should be chosen to have a simple and efficient implementation in many machines, to avoid execution costs for unneeded generality, and to ensure that unused portions of a CAIS implementation will not add to execution costs of a non-using tool. The measures of the efficiency criterion are, primarily, minimum interactive response time for APSE tools and, secondarily, consumption of resources.

The PCTE meets this objective. Regarding efficiency, an analysis of the Bull PCTE and Emeraude entity management support leads us to believe its efficiency will be comparable to that of equivalent UNIX facilities.

2.4 Modularity.

The RAC requires that interfaces should be partitioned such that the partitions may be understood independently and they contain no undocumented dependencies between partitions.

PCTE is partitioned in a UNIX style. There are dependencies between the partitions, particularly as they relate to extending the underlying UNIX functions with distributed network communications, and the entity-relationship model. If the interfaces were respecified in Ada, it would be possible to devise a scheme of package structuring with fewer dependencies, although some dependencies between interfaces are inevitable in an operating system. (See also the comment to section 3.2E.)

2.5 Extensibility.

The RAC requires that the design of the CAIS should facilitate development and use of extensions of the CAIS; i.e., CAIS interfaces should be reusable so that they can be combined to create new interfaces and facilities.

All PCTE interfaces are program callable and may therefore be extended using the facilities of the programming language.

2.6 Technology Compatibility.

The RAC requires that the CAIS shall adopt existing standards where applicable. For example the RAC recognizes standards by ANSI, ISO, IEEE, and DoD.

PCTE's goal is conformance with applicable international (ISO) standards. An Ada version of the PCTE interfaces would also conform to international standards. Generally these intersect with and conform to the applicable ANSI and IEEE standards. Conformance with DoD standards is unlikely at this time, because applicable DoD standards have not been called out.

2.7 Uniformity.

The RAC requires that the CAIS features should uniformly address aspects such as status returns, exceptional conditions, parameter types, and options. Different modules within the CAIS should be specified to the same logical level, and a small number of unifying conceptual models should underlie the CAIS.

The PCTE (C interfaces) inherit the UNIX style of uniform treatment of status returns, signal handling, parameter types and options. PCTE, as a set of Ada interfaces, should meet the RAC requirement, but not necessarily in the same manner as the current C version of PCTE.

The main "conceptual models" in PCTE are:

- *objects, attributes, and links,*
- *the notion of schema,*
- *the notion of process and "transaction",*
- *the notion of "emissary" to implement distribution, and*
- *the unified notion of intercommunicating agents at the user-interface level.*

2.8 Security.

The RAC requires that the CAIS shall provide interfaces to allow tools to operate within a Trusted Computer System (TCS) that meets the Class B3 criteria as defined in The U.S. DoD Computer Security Center, *Trusted Computer System Evaluation Criteria*, CSC-STD-001-83. Specifically:

a. It shall be possible to implement the CAIS within a TCS.

b. When implemented within a TCS, the CAIS shall support the use of the security facilities provided by the Trusted Computing Base (TCB) to applications programs.

c. When not implemented within a TCS, the CAIS interfaces sensitive to security shall operate as a dedicated secure system (i.e., all data at a single security level, and all subjects cleared to at least that level).

The PCTE has not been compared to the Class B3 criteria. In general, its authors have been sponsored to produce a commercial/industrial product, and B3 criteria may impose additional requirements.

3. GENERAL SYNTAX AND SEMANTICS

3.1 Syntax

Section	Conformance	
	requirement	RAC to PCTE
3.1A General Syntax	shall	In progress
3.1B Uniformity	should	In progress
3.1C Name Selection	should	In progress
3.1D Pragmatics	should	Yes

3.1A General Syntax. The RAC requires that the syntax of the CAIS shall be expressed as Ada package specifications. The syntax of the CAIS shall conform to the character set as defined by the Ada standard (section 2.1 of ANSI/MIL-STD-1815A).

The Ada specifications of PCTE are in preparation. They will, no doubt, meet this requirement.

3.1B Uniformity. The RAC requires that the CAIS should employ uniform syntactic conventions and should not provide several notations for the same concept. CAIS syntax issues (including, at least, limits on name lengths, abbreviation styles, other naming conventions, relative ordering of input and output parameters, etc.) should be resolved in a uniform and integrated manner for the whole CAIS.

PCTE's C interfaces retain compatibility to UNIX interfaces, because of a requirement for binary code compatibility to pre-existing UNIX programs. This has caused several of the "old style" UNIX interfaces, which are preserved in PCTE, to provide a "duplicate notation" to the newer and more robust PCTE interfaces. Additionally, because of the UNIX-level requirement to keep interface names generally shorter than seven characters, many awkward notations result.

Furthermore, the C interfaces with PCTE do not have Ada's overloaded names resolution facility. This results in multiple notations for the same service where parametric alternatives exist.

A PCTE implementation of Ada interfaces, particularly if these conformed to the CAIS, would resolve many of the notational inconsistencies. An Ada interface could hide the underlying C-named interfaces (though there has been some argument of adopting a different granularity in PCTE Ada vs PCTE C specified interfaces).

Given that the translation of PCTE's C interfaces into Ada resolves these issues as assumed in section 1.3, then conformance is provided. If, however, the current PCTE interfaces are simply transliterated, then this RAC requirement is not met.

3.1C Name Selection. The RAC requires that the CAIS should avoid coining new words (literals or identifiers) and should avoid using words in an unconventional sense. Ada identifiers (names) defined by the CAIS should be natural language words or industry accepted terms whenever possible. The CAIS should define Ada identifiers which are visually distinct and not easily confused (including, at least, that the CAIS should avoid defining two Ada identifiers that are only a 2-character transposition away from being identical). The CAIS should use the same name everywhere in the interface set, and not its possible synonyms, when the same meaning is intended.

See preceding discussion. The PCTE C implementation contains a great deal of unconventional words and terms, due to conformity to the UNIX environment. An Ada implementation (without the requirement to have compatibility for preexisting programs) is assumed to provide a new set of names and terms, to use a good Ada "style".

3.1D Pragmatics. The RAC requires that the CAIS should impose only those restrictive rules or constraints required to achieve I&T. CAIS implementors will be required to provide the complete specifications of all syntactic restrictions imposed by their CAIS implementations.

The PCTE meets this requirement.

3.2 Semantics

Section	Conformance	
	requirement	RAC to PCTE
3.2A General Semantics	shall	Yes
3.2B Responses	shall provide	Yes
3.2C Exceptions	shall	Partial
3.2D Consistency	should	Probably yes
3.2E Cohesiveness	should	Reasonable
3.2F Pragmatics	shall	Yes

3.2A General Semantics. The RAC requires that the CAIS shall be completely and unambiguously defined. The specification of semantics should be both precise and understandable. The semantic specification of each CAIS interface shall include a precise statement of ass mptions (including execution-time preconditions for calls), effects on global data and packages, and interactions with other interfaces.

The PCTE is defined as completely and unambiguously as CAIS 1.

We feel, however, that an informal form of specifications, like that adopted by PCTE or by CAIS 1, cannot be adequately precise. A definition using some formal means of description is needed, at least as guidance for implementors.

There is some reason to hope that a formal definition of PCTE will be undertaken in the near future.

3.2B Responses. The RAC requires that the CAIS shall provide responses for all interface calls, including informative non-null responses (return value or exception) for unsuccessful completions. All responses returned across CAIS interfaces shall be defined in an implementation-independent manner. Every time a CAIS interface is called under the same circumstances, it should return the same response.

The C version of the PCTE meets this requirement, and any Ada versions most certainly would too.

3.2C Exceptions. The CAIS interfaces shall employ the mechanism of Ada exceptions to report exceptional situations that arise in the execution of CAIS facilities. The CAIS specification shall include exceptions (with visible declarations) for all situations that violate the preconditions specified for the CAIS interfaces. The CAIS specification shall include exceptions (with visible declarations) that cover all violations of implementation-defined restrictions.

In general, a PCTE Ada interface, implemented as per the assumption with good Ada style, should meet this requirement. However, there are certain underlying UNIX implementation issues which might "leak through" to such an implementation. For example, UNIX permits excessively long file and identifier names and truncates their length, rather than providing the equivalent of CAIS 1's USE_ERROR. This inconsistency can be resolved at the Ada library routine interface, by the incorporation of additional checking.

3.2D Consistency. The description of CAIS semantics should use the same word or phrase everywhere, and not its possible synonyms, when the same meaning is intended.

A thorough analysis of PCTE, in search of violations, has not been conducted.

3.2E Cohesiveness. Each CAIS interface should provide only one function.

This is a goal which is limited by the desired properties of operating system implementations. Most kernel level functions provide multiple functions, especially when they effect the stored value of an object, a transaction status, and another executing process. We believe that the PCTE *functionality, and accompanying side effects, is at least as cohesive as other modern operating systems.*

3.2F Pragmatics. The CAIS specification shall enumerate all aspects of the meanings of CAIS interfaces and facilities which must be defined by CAIS implementors. CAIS implementors will be required to provide the complete specifications for these implementation-defined semantics.

PCTE meets this requirement.

4. ENTITY MANAGEMENT SUPPORT

Section	Conformance	
	requirement	RAC to PCTE
Access controls & security	will apply	Partial
a. means for retaining data	shall	Yes
b. relationships and properties of data	shall	Yes
c. operating upon data	shall	Yes
d. defining & enforcing legality definitions	shall	Yes
e. descriptions of data	shall	Yes
f. inheriting properties of definitions	shall	Yes
g. relationships separate from data instances	shall	Yes
h. data separate from tools	shall	Yes

The RAC states that access controls and security rights will apply to all CAIS facilities required in this section.

PCTE provides access controls and security rights according to a UNIX model (see Section 2.8).

The RAC states that the general requirements for the CAIS entity management support are the following.

a. There shall be a means for retaining data.

Provided.

b. There shall be a way for retaining relationships among and properties of data.

Provided.

c. There shall be a way of operating upon data, deleting data, and creating new data.

Provided.

d. There shall be a means for defining certain operations and conditions as legal, for enforcing the definitions, and for accepting additional definitions of legality.

Provided.

e. There shall be a means to describe data, and there shall be a means to operate upon such descriptions. Descriptions of the data shall be distinguished from the data described.

Provided.

f. There shall be a way to develop new data descriptions by inheriting (some of) the properties of existing data descriptions.

Provided.

g. The relationships and properties of data shall be separate from the existence of the data instances.

Provided

h. The descriptions of data and the instances of data shall be separate from the tools that operate upon them.

Provided.

The RAC characterization (subsections 4.1 - 4.7) of Entity Management Support is based on the STONEMAN requirements for a database, using a model based on the entity-relationship concept. Although a CAIS design meeting these requirements is expected to demonstrate the characteristics and capabilities reflected here, it is not necessary that such a design directly employ this entity-relationship model.

The RAC entity-relationship model, for which definitions and requirements follow in 4.1 - 4.7, fulfills these requirements, and any alternative data model shall fulfill these requirements and shall also fulfill the equivalent of the requirements in 4.1 through 4.7.

4.1 Entities, Relationships, and Attributes.

Section	Conformance	
	requirement	RAC to PCTE
4.1A Data	shall provide	Yes
4.1B Elementary values	shall provide	Yes
4.1C System Integrity	shall	Yes
4.2A Types	shall	Yes (see text)
4.2B Type Rules	shall	Yes (see text)
4.2C Type definitions	shall provide	Yes
4.2D Changing type definitions	shall provide	Yes
4.2E Triggering	shall provide	No
4.3A Exact Identities	shall provide	No
4.3B Identification	shall provide	Partial
4.3C Identification methods	shall provide	Partial

The following RAC definitions, used in this subsection, pertain to all the rest of section 4 also:

entity a representation of a person, place, event or thing.

relationship an ordered connection or association among entities. A relationship among N entities (not necessarily distinct) is known as an "N-ary" relationship.

attribute an association of an entity or relationship with an elementary value.

elementary value one of two kinds of representations of data: interpreted and uninterpreted.

interpreted data a data representation whose structure is controlled by CAIS facilities and may be used in the CAIS operations. According to the RAC, examples are representations of integer, string, real, date and enumeration data, and aggregates of such data.

According to the C definition of the PCTE, the four allowed representations are integer, boolean, date, or string.

uninterpreted data a data representation whose structure is not controlled by CAIS facilities and whose structure is not used in the CAIS operations. Examples might be representations of files, such as requirements documents, program source code, and program object code.

4.1A **Data.** The RAC requires that the CAIS shall provide facilities for representing data using entities, attributes or binary relationships. The CAIS may provide facilities for more general N-ary relationships, but it is not required to do so.

PCTE provides these facilities.

4.1B **Elementary Values.** The RAC requires that the CAIS shall provide facilities for representing data as elementary values.

Provided: PCTE provides both interpreted data ("attributes") and uninterpreted data ("files"); however, it does not provide all of the examples enumerated above or aggregates as interpreted data. Aggregates could be implemented by Ada interfacing library routines, and then stored in other representation forms. [Conformance is listed as "yes" because no statement is made as to a precise list of mandatory representation forms.]

4.1C **System Integrity.** The CAIS facilities shall ensure the integrity of the CAIS-managed data.

The PCTE appears to provide these facilities.

4.2 Typing.

The following RAC definition, used in this subsection, pertains to all the rest of section 4 also:

typing an organization of entities, relationships and attributes in which they are partitioned into sets, called entity types, relationship types and attribute types, according to designated type definitions.

4.2A Types. The RAC requires that the facilities provided by the CAIS shall enforce typing by providing that all operations conform to the type definitions. Every entity, relationship and attribute shall have one and only one type.

Provided: PCTE has a specific model of types.

- *Entities of PCTE are typed in that their partitioning into sets (to use RAC terminoiogy) restricts their set of allowed attributes and their set of allowed link types emanating from the type (set of) entities. Every entity can have only one type.*
- *Relationships of PCTE are also typed; this restricts their allowed set of attributes, the (set of) key attributes, the set of allowed destination entities, and the "arity" of the link. A link may be of only one type.*
- *Attributes in the C version of PCTE are typed; this restricts their value types, and their initial values. (Attributes are applied to various entity or relationship types in the PCTE model. Unlike CAIS 1, once a given attribute is "applied" to a relationship type, it exists for all relationships of that type. (It doesn't occupy storage, however, if its value is the default.)*

  In the C version, the representation types are limited to integer, boolean, date, or string. An Ada version of PCTE might provide Ada typing, and thus facilitate generalized attribute representations, aggregates, and the like.

In this situation, the PCTE provides a model for meeting the RAC requirements which goes significantly beyond CAIS 1.

4.2B Rules about Type Definitions. The RAC requires that the CAIS type definitions shall:

- Specify the entity types and relationship types to which each attribute type may apply.

  PCTE does this.

- Specify the type or types of entities that each relationship type may connect and the attribute types allowed for each relationship type.

  PCTE does this by specifying, for an entity, the allowed emanating relationships; for a relationship, the allowed destination entities; and for each entity and relationship type, the applied attribute types.

- Specify the set of allowable elementary values for each attribute type.

  The allowable values of these are the basic types.

- Specify the relationship types and attribute types for each entity type.

  As noted above, with PCTE, for each entity type, emanating relationship types are controlled. Destination relationship types are only controlled because the relationship type itself restricts the allowed destinations. This model may be slightly different from that of the RAC, but we assume it is functionally equivalent.

  PCTE requires the schema to specify which attribute types are applied to each entity type.

- Permit relationship types that represent either functional mappings (one-to-one or many-to-one) or relational mappings (one-to-many or many-to-many).

PCTE provides all of these relationship types.

- Permit multiple distinct relationships among the same entities.

  PCTE permits this.

- Impose a lattice structure on the types which includes inheritance of attributes, attribute value ranges (possibly restricted), relationships and allowed operations.

  PCTE only provides tree structures. See above for attribute value range discussion.

4.2C **Type Definition.** The RAC requires that the CAIS shall provide facilities for defining new entity, relationship and attribute types.

PCTE provides this.

4.2D **Changing Type Definitions.** The RAC requires that the CAIS shall provide facilities for changing type definitions. These facilities shall be controlled such that data integrity is maintained.

PCTE meets this requirement.

4.2E **Triggering.** The RAC requires that the CAIS shall provide a conditional triggering mechanism so that prespecified procedures or operations (such as special validation techniques employing multiple attribute value checking) may be invoked whenever values of indicated attributes change. The CAIS shall provide facilities for defining such triggers and the operations or procedures which are to be invoked.

PCTE, in the C version, does not provide such a facility. [We doubt that this facility would be provided by an Ada version of PCTE, either.]

Implementation of triggering would require PCTE kernel changes. A facility could be implemented for an enhanced PCTE kernel, where attribute triggering were supported by schema entries, effectively and securely enforced for both Ada and non-Ada usage of the attributes.

4.3 Identification

The following RAC definitions, used in this subsection, pertain to all the rest of section 4 also:

exact identity a designation of an entity (or relationship) that is always associated with the entity (or relationship) that it designates. This exact identity will always designate exactly the same entity (or relationship), and it cannot be changed.

identification a means of specifying the entities, relationships and attributes to be operated on by a designated operation.

4.3A **Exact Identities.** The RAC requires that the CAIS shall provide exact identities for all entities. The CAIS shall support exact identities for all relationships. The exact identity shall be unique within an instance of a CAIS implementation, and the CAIS shall support a mechanism for the utilization of exact identities across all CAIS implementations.

PCTE does not provide exact identities for the entities of its object management system. It would not be difficult to modify PCTE to include such a facility; a composition of the initial IVOL, INO, and creation date attribute would be sufficient for this. For relationships the exact identity could be supported by identifying the exact identity of the source entity and the key of the link. We feel that with such minor internal PCTE changes, this requirement could be fully met.

A particularly valuable ability of PCTE is the ability of PCTE relationships to track entity movement in a distributed system.

A CAIS 2 implementation superimposed on a PCTE kernel could implement the required mechanism, though it would then not be available to non-Ada programs.

4.3B Identification. The RAC requires that the CAIS shall provide identification of all entities, attributes and relationships. The CAIS shall provide identification of all entities by their exact identity. The RAC requires that the CAIS shall support identification of all relationships by their exact identity.

PCTE meets this requirement except in the case of identification by exact identity.

4.3C Identification Methods. The RAC requires that the CAIS shall provide identification of entities and relationships by at least the following methods:

- Identification of some "start" entity(s), the specification of some relationship type and the specification of some predicate involving attributes or attribute types associated with that relationship type or with some entity type. This method shall identify those entities which are related to the identified start entity(s) by relationships of the given relationship type and for which the predicate is true. Subject to the security constraints of section 2.8, all relationships and entities shall be capable of identification via this method, and all attributes and attribute types (except uninterpreted data) shall be permitted in the predicates.

  PCTE only allows those attributes defined to be part of the relationship key to be involved in the predicate. Because this requirement specifies that a "predicate" permits any or all attributes and types to be in a predicate expression (except uninterpreted data), PCTE does not meet this.

  This requirement could be supported by an interpretive implementation of predicate processing in the PCTE kernel, or in a library routine outside the kernel. This generalized form of predicate processing is likely to have performance limitations in any CAIS implementation.

- Identification of an entity type or relationship type and specification of some predicate on the value of any attribute of the entity type or relationship type. This method shall identify those entities or relationships of the given type for which the predicate is true. Subject to the security constraints of section 2.8, all attributes (except uninterpreted data) shall be permitted in the predicates.

  PCTE does not provide an efficient mechanism to support this requirement. It is possible, however, to satisfy it by search of the entire database. In any CAIS implementation, this is likely to impose even more severe performance problems than the preceding form of identification.

4.4 Operations

Section	Conformance	
	requirement	RAC to PCTE
4.4A Entity Operations	shall provide	Yes (with caveats)
4.4B Relationship Operations	shall provide	Yes (with caveats)
4.4C Attribute Operations	shall provide	Yes
4.4D Exact Identity Operations	shall provide	Inapplicable
4.4E Uninterpreted Data Operations	shall provide	Yes - see assumption
4.4F Synchronization	shall provide	Yes
4.4G Access Control	shall provide	Partial
4.5A Transaction Mechanism	shall support	Yes (provided)
4.5B Transaction Control	shall support	Yes (provided)
4.5C System Failure	shall	Yes
4.6A History Mechanism	shall support	Yes
4.6B History Integrity	shall support	Yes
4.7A Robustness & Restoration	shall support	Yes (with caveats)

4.4A Entity Operations. The RAC requires that the CAIS shall provide facilities to:

- create entities

- delete entities
- examine entities (by examining their attributes and relationships)
- modify entities (by modifying their attributes)
- identify entities (as specified in Section 4.3)

The PCTE meets this requirement with the caveat that source identification methods of section 4.3 cannot be supported. It should also be noted that deletion of an entity is achieved by deletion of composition links to the entity, rather than a single operation of deletion.

4.4B Relationship Operations. The RAC requires that the CAIS shall provide facilities to:

- create relationships
- delete relationships
- examine relationships (by examining their attributes)
- modify relationships (by modifying their attributes)
- identify relationships (as specified in Section 4.3)

The PCTE meets this requirement with the caveat that some identification methods of section 4.3 cannot be supported.

4.4C Attribute Operations. The RAC requires that the CAIS shall provide facilities to:

- examine attributes
- modify attributes

The PCTE meets this requirement.

4.4D Exact Identity Operations. The RAC requires that the CAIS shall provide facilities to:

- pass exact identities between processes
- compare exact identities

In the absence of exact identities in PCTE this requirement in inapplicable (see section 4.3A).

4.4E Uninterpreted Data Operations. The RAC requires that the CAIS shall provide that use of the input-output facilities of the Ada language (as defined in Chapter 14 of ANSI/MIL-STD-1815A) results in reading/writing an uninterpreted data attribute of an entity. The facilities of Section 6 shall then apply.

The PCTE meets this requirement with the assumption that an Ada language implementation is provided which provides the Ada facilities as above. Since the uninterpreted data attribute of an entity is the equivalent of the UNIX "file", it should be expected of Ada implementations to meet this requirement.

4.4F Synchronization. The RAC requires that the CAIS shall provide dynamic access synchronization mechanisms to individual entities, relationships and attributes.

The PCTE provides these features in the Concurrency and Integrity Control Facilities functions.

4.4G Access Control. The RAC requires that the CAIS shall provide selective prohibition of operations on entities, relationships, and attributes being requested by an individual.

This requirement appears to imply discretionary access control; PCTE defines access control according to a UNIX model, and does not selectively distinguish between access prohibitions to specific attributes and not others for given entities or relationships. For this reason the compliance is noted as "partial".

CAIS 1 supports a somewhat finer level of access control. If CAIS 2 supports this same level then PCTE will require modification in order to ensure convergence. It should be noted that fine granule access rights may have performance consequences.

4.5 Transaction

The following definition, used in this subsection, pertains to all the rest of section 4 also:

transaction a grouping of operations, including a designated sequence of operations, which requires that either all of the designated operations are applied or none are; e.g., a transaction is uninterruptible from the user's point of view.

4.5A Transaction Mechanism. The RAC requires that the CAIS shall support a transaction mechanism. The effect of running transactions concurrently shall be as if the concurrent transactions were run serially.

Though only required to "support," PCTE "provides" this facility. In addition a user can choose the level of transaction protection.

4.5B Transaction Control. The RAC requires that the CAIS shall support facilities to start, end and abort transactions. When a transaction is aborted, all effects of the designated sequence of operations shall be as if the sequence were never started.

Though only required to "support," PCTE "provides" this facility.

4.5C System Failure. System failure while a transaction is in progress shall cause the effects of the designated sequence of operations to be as if the sequence were never started.

PCTE provides this facility. With UNIX a utility runs after system failure to clean up the filesystem. In PCTE the corresponding utility also "undoes" the effect of partially completed transactions and of transactions which completed but weren't fully committed when the system failed.

4.6 History

The following RAC definitions, used in this subsection, pertain to all the rest of section 4 also:

history a recording of the manner in which entities, relationships and attribute values were produced and of all information which was relevant in the production of those entities, relationships or attribute values.

4.6A History Mechanism. The RAC requires that the CAIS shall support a mechanism for collecting and utilizing history. The history mechanism shall provide sufficient information to support comprehensive configuration control.

PCTE "supports" such a mechanism. It provides sufficient entity attributes for the mechanism to be implemented. There are a large variety of history mechanisms implemented on UNIX, and the facilities they use are provided by PCTE.

4.6B History Integrity. The RAC requires that the CAIS shall support mechanisms for ensuring the fidelity of the history.

History is stored as data in PCTE. General mechanisms for ensuring the stability of data are provided.

4.7 Robustness and Restoration

The following RAC definitions, used in this subsection, pertain to all the rest of section 4 also:

backup a redundant copy of some subset of the CAIS-managed data. The subset is capable of restoration to active use by a CAIS implementation, particularly in the event of a loss of completeness or integrity in the data in use by implementation.

archive a subset of the CAIS-managed data that has been relegated to backing storage media while retaining the integrity, consistency and availability of all information in the entity management system.

4.7A **Robustness and Restoration.** The RAC requires that the CAIS shall support facilities which ensure the robustness of and ability to restore CAIS-managed data. The facilities shall include at least those required to support the backup and archiving capabilities provided by modern operating systems.

Backup could be supported in a PCTE implementation by volume dump and restore, but this might not preserve full integrity. It is an open question as to whether fully consistent backup can be achieved in an economic way in a distributed system which allows disconnection of workstations in the sense of PCTE.

Archiving is supported by moving data to a mountable volume which is then removed. This preserves full consisteney and integrity of the data.

PCTE also provides a variety of other features which may be used in a similar way to UNIX to provide analogous features to UNIX, but the means of preserving integrity are unclear.

5. PROGRAM EXECUTION FACILITIES

Section	Conformance	
	requirement	RAC to PCTE
Access controls & security	will apply	Partial
5.1A Activation	shall provide	Yes
5.1B Unambiguous Identification	shall provide	Yes
5.1C Activation Data	shall provide	Yes
5.1D Dependent Activation	shall provide	Yes (see text)
5.1E Independent Activation	shall provide	Yes (see text)
5.2A Termination	shall provide	Yes
5.2B Termination of Dependent Process	shall support	Yes
5.2C Termination Data	shall provide	Partial (see text)
5.3A Data Exchange	shall provide	Yes
5.4A Task Waiting	shall support	Yes (see text)
5.4B Parallel Execution	shall provide	Yes
5.4C Synchronization	shall provide	Yes
5.4D Suspension	shall provide	Yes
5.4E Resumption	shall provide	Yes
5.5A Identity Reference	shall provide	Yes
5.5B RTS Independence	shall	Yes
5.5C Instrumentation	shall provide	Yes

The RAC states that access controls and security rights will apply to all CAIS facilities required by this section.

PCTE provides access controls on the static context of a program (e.g., the file storing the executable code), but inherits the UNIX rules of access to dynamic contexts (the process representation):

- *The "ps" command allows any user to enquire about all other user's running processes (either by exact identity or limited predicate).*
- *The "kill" command allows any user to enquire about the existence of another running process by exact identity (PID).*

Therefore, the compliance to this requirement is considered "partial" for PCTE.

PCTE could provide significant additional access control by kernel extension.

The following RAC definitions are used in this section:

process the CAIS facility used to represent the execution of any program.

program a set of compilation units, one of which is a subprogram called the "main program." Execution of the program consists of execution of the main program, which may invoke subprograms declared in the compilation units of the program.

resource any capacity which must be scheduled, assigned, or controlled by the operating system to assure consistent and non-conflicting usage by programs under execution. Examples of resources include: CPU time, memory space (actuals and virtual), and shared facilities (variables, devices, spoolers, etc.).

activate to create a CAIS process. The activation of a program binds that program to its execution environment, which are the resources required to support the process's execution, and includes the program to be executed. The activation of a process marks the earliest point in time which that process can be referenced as an entity within the CAIS environment.

terminate to stop the execution of a process such that it cannot be resumed.

deactivate to remove a terminated process so that it may no longer be referenced within the CAIS environment.

suspend to stop the execution of a process such that it can resumed. In the context of an Ada program being executed, this implies the suspension of all tasks, and the prevention of the activation of any task until the process is resumed. It specifically does not imply the release of any resources which a process has assigned to it, or which it has acquired, to support its execution.

resume to resume the execution of a suspended process.

task wait delay of the execution of a task within a process until a CAIS service requested by this task has been performed. Other tasks in the same process are not delayed.

5.1 Activation of Program

5.1A Activation. The RAC requires that the CAIS shall provide a facility for a process to create a process for a program that has been made ready for execution. This event is called activation.

PCTE provides several facilities which accomplish activation.

5.1B Unambiguous Identification. The RAC requires that the CAIS shall provide facilities for the unambiguous identification of a process at any time between its activation and deactivation; one such capability shall be as an indivisible part of activation. This act of identification establishes a reference to that process. Once such a reference is established, that reference will refer to the same process until the reference is dissolved. A reference is always dissolved upon termination of the process that established the reference. A terminated process may not be deactivated while there are references to that process.

PCTE provides a process identifier (PID), which is valid between activation and deactivation. This meets the RAC requirement.

CAIS 1 has a process identifier model closely integrated with the node identifier model. This is quite different from the PCTE process identifier model. If, as seems possible, the CAIS 2 model is similar to CAIS 1 then changes to PCTE might be needed to achieve convergence.

5.1C Activation Data. The RAC requires that the CAIS shall provide a facility to make data available to a program upon its activation.

PCTE provides such a facility.

5.1D Dependent Activation. The RAC requires that the CAIS shall provide a facility for the activation of programs that depend upon the activating process for their existence.

This requirement is not precise. The following PCTE features appear to satisfy reasonable interpretations of this requirement:

[Note: In the following, START, END and ABORT indicate events which occur in response to calling specific PCTE interface routines with those (or similar) names.]

In PCTE,

a. *selected processes can be associated with an "activity" (which can be considered a dynamic context in which the selected processes execute); the process which STARTS an activity is said to "act in behalf of the activity" ard is an ancestor (in the UNIX sense) of any other processes in the activity. When such a "starter" task ENDS normally and when all other processes in the activity have terminated, then all the processes in the activity are deactivated. When such a "starter" process is ABORTed, a signal is sent to the other processes in the activity to encourage their termination. When all those processes have terminated, then all the processes in the activity are deactivated (just as in the "END normally" case).*

b. *processes not associated with an activity are activated, terminated and deactivated according to the rules of UNIX.*

5.1E Independent Activation. The RAC requires that the CAIS shall provide a facility for the activation of programs that do not depend upon the activating process for their existence.

The PCTE appears to comply with this requirement. The RAC is not precise here. It is possible to set up a PCTE child process so that it can continue after its parent has been terminated.

5.2 Termination

5.2A Termination. The RAC requires that the CAIS shall provide a facility for a process to terminate a process. There shall be two forms of termination; the voluntary termination of a process (termed completion) and the abnormal termination of a process. Completion of a process is always self-determined, whereas abnormal termination may be initiated by other processes.

PCTE provides both of these forms of termination.

5.2B Termination of Dependent Processes. The RAC requires that the CAIS shall support clear, consistent rules defining the termination behavior of processes dependent on a terminating process.

PCTE meets this requirement. See 5.1D and 5.1E.

5.2C Termination Data. The RAC requires that the CAIS shall provide a facility for termination data to be made available. This data shall provide at least an indication of success or failure for processes that complete. For processes that terminate abnormally the termination data shall indicate abnormal termination.

PCTE, under normal circumstances, only makes termination data available to the parent of a terminated but not yet deactivated process. The act of reading this data makes the process deactivated, which means it is no longer available. This is perhaps not in the spirit of the RAC.

5.3 Communication

5.3A Data Exchange. The RAC requires that the CAIS shall provide a facility for the exchange of data among processes.

PCTE provides several such facilities; these include messages, pipes, and shared memory.

5.4 Synchronization

5.4A Task Waiting. The RAC requires that the CAIS shall support task waiting.

Task waiting is not "provided" by PCTE — calls to the PCTE interfaces are, in general, blocking on the whole process. Task waiting could be "supported" for a customer process by interprocess communications to agent processes which carry out the requested PCTE calls on behalf of the customer process.

(The Ada Run-Time System (RTS) has to be designed to cooperate with this mechanism.)

5.4B Parallel Execution. The RAC requires that the CAIS shall provide for the parallel execution of processes.

PCTE provides for this.

5.4C Synchronization. The RAC requires that the CAIS shall provide a facility for the synchronization of cooperating processes.

PCTE provides signals, messages, waits and locks, which provide this facility.

5.4D Suspension. The RAC requires that the CAIS shall provide a facility for suspending a process.

PCTE provides this facility.

5.4E Resumption. The RAC requires that the CAIS shall provide a facility to resume a process that has been suspended.

PCTE provides this facility.

5.5 Monitoring

5.5A Identity Reference. The RAC requires that the CAIS shall provide a facility for a process to determine an unambiguous identity of a process and to reference that process using that identity.

PCTE provides this facility.

5.5B RTS Independence. CAIS program execution facilities shall be designed to require no additional functionality in the RTS from that provided by Ada semantics. Consequently, the implementation of the Ada RTS shall be independent of the CAIS.

We strongly believe that this requirement contradicts with 5.4A, the Task Waiting requirement.

PCTE intends to provide full binary code compatibility with standard UNIX. This would allow an Ada version of PCTE to support UNIX-compatible Ada compilers which have RTS's independent of the CAIS.

5.5C Instrumentation. The RAC requires that the CAIS shall provide a facility for a process to inspect and modify the execution environment of another process. This facility is intended to promote support for portable debuggers and other instrumentation tools.

PCTE provides this facility. It can, however only be used on processes which are children of the inspector and modifier process.

6. INPUT/OUTPUT

[This section contains several answers based on UNIX knowledge of System III, extrapolated to assumed implementation by System V.]

Section	Conformance	
	requirement	RAC to PCTE
Access controls & security	will apply	Partial
6.1A Hardcopy Terminals	shall provide	Yes
6.1B Page Terminals	shall provide	Yes
6.1C Printers	shall provide	Yes
6.1D Paper Tape Drives	shall provide	Yes
6.1E Graphics Support	shall support	Yes
6.1F Telecommunications Support	shall support	Yes
6.2A Block Terminals	shall provide	Unknown
6.2B Tape Drives	shall provide	Unknown

The RAC requires that access controls and security rights will apply to all CAIS facilities required by this section.

The PCTE provides the same access controls and security for the devices specified in this section as it does for entity management support. (See Section 2.8.)

The RAC states that the requirements specified in this section pertain to input/output between/among objects (e.g. processes, data entities, communication devices, and storage devices) unless otherwise stated. All facilities specified in the following requirements are to be available to non-privileged processes, unless otherwise specified.

The following RAC definitions will be used in this section:

block terminal a terminal that transmits/receives a block of data units at a time.

consumer an entity that is receiving data units via a datapath.

data block a sequence of one or more data units which is treated as an indivisible group by a transmission mechanism.

data unit a representation of a value of an Ada discrete type.

datapath the mechanism by which data units are transmitted from a producer to a consumer.

datastream the data units flowing from a producer to a consumer (without regard to the implementing mechanism).

hardcopy terminal a terminal which transmits/receives one data unit at at time and does not have an addressable cursor.

page terminal a terminal which transmits/receives one data unit

producer an entity that is transmitting data units via a datapath.

terminal an interactive input/output device.

type-ahead the ability of a producer to transmit data units before the consumer requests the data units

6.1 Virtual I/O Devices: Data Unit Transmission

6.1A Hardcopy Terminals. The RAC requires that the CAIS shall provide interfaces for the control of hardcopy terminals.

PCTE provides for conventional UNIX support of these devices. This includes the use of the CURSES *and* TERMCAP *facilities, which provide device control by subroutine call, rather than via device-specific data units.*

6.1B Page Terminals. The RAC requires that the CAIS shall provide interfaces for the control of page terminals.

See 6.1A

6.1C Printers. The RAC requires that the CAIS shall provide interfaces for the control of character-imaging printers and bit-map printers.

See 6.1A

6.1D Paper Tape Drives. The RAC requires that the CAIS shall provide interfaces for the control of paper tape drives.

The specific implementation of Paper Tape Devices is not specified by PCTE or UNIX. Common practice is for vendors of interfaces and drivers to supply accompanying software. Interfaces for their control typically rely on the UNIX "ioctl" function, to avoid use of embedded datastream control sequences.

6.1E Graphics Support. The RAC requires that the CAIS shall support the control of interactive graphical input/output devices. [The RAC appears to not distinguish between graphical input devices, which usually are raster imaging devices, and locator devices, such as mice. We presume the intent is to require locator, rather than raster imaging, input.]

PCTE defines an extensive set of User Interface primitives, which deal with graphical display (output) devices. It also defines locator device input primitives.

6.1F Telecommunications Support. The RAC requires that the CAIS shall support a telecommunications interface for data transmission.

The PCTE functional description refers "ioctl" (and associated commands) to the UNIX System V description. These features of UNIX provide for extensive telecommunications support for data transmission.

6.2 Virtual I/O Devices: Data Block Transmission

6.2A Block Terminals. The RAC requires that the CAIS shall provide interfaces for the control of character-imaging block terminals.

PCTE support for these devices is unknown at this time. However, it is likely that the interfaces provided by low level input output control, and by higher level CURSES *and* TERMCAP *facilities could either be adapted to, or already do support these forms of terminals.*

6.2B Tape Drives. The RAC requires that the CAIS shall provide interfaces for the control of magnetic tape drives.

PCTE support for tape drives is unknown at this time. However, it is likely that the interfaces provided by low level input output control, and by UNIX utilities provided in System V, are sufficient to meet this requirement.

6.3 Datapath Control

Section	Conformance	
	requirement	RAC to PCTE
6.3A Datapath Interface Level	shall provide	Yes
6.3B Datapath Timeout	shall provide	Partial
6.3C Exclusive Access	shall provide	Yes
6.3D Datastream Redirection	shall provide	Yes
6.3E Datapath Buffer Size	shall provide	Yes - see text
3.3F Datapath Flushing	shall provide	Yes
6.3G Output Datapath Processing	shall provide	Yes
6.3H Input/Output Sequencing	shall provide	Unknown

This section discusses RAC requirements and criteria which pertain to both data unit transmission and block transmission.

6.3A **Interface Level.** The RAC requires that datapath control facilities of the CAIS shall be provided at a level comparable to that of Ada Reference Manual's File I/O. That is, control of datapaths shall be provided via subprogram calls rather than via the data units transmitted to the device.

PCTE provides the appropriate datapath control facilities subroutine calls, as described below.

6.3B **Timeout.** The RAC requires that the CAIS shall provide facilities to permit timeout on input and output operations.

PCTE provides partial support for this requirement. The "alarm" function and the ability to catch it's signals can implement timeouts; support is listed, however, as partial because a process may have a number of concurrent input/output operations and wish to implement timeouts for each of them, and PCTE supports only one alarm timer for a process. Thus a library routine implementation to schedule and arbitrate the alarm's would be required.

A CAIS 2 implementation, if it follows the CAIS 1 model of providing timeouts on CAIS calls, would implement alarm arbitration within the library modules which interface the CAIS 2 to the underlying PCTE system.

6.3C **Exclusive Access.** The RAC requires that the CAIS shall provide facilities to obtain exclusive access to a producer/consumer; such exclusive access does not prevent a privileged process from transmitting to the consumer.

PCTE provides the same level of access restrictions for devices as it provides for the entity management facilities. See 4.4G.

6.3D **Datastream Redirection.** The RAC requires that the CAIS shall provide facilities to associate at execution time the producer/consumer of each input/output datastream with a specific device, data entity, or process

PCTE provides these facilities.

6.3E **Datapath Buffer Size.** The RAC requires that the CAIS shall provide facilities for the specification of the sizes of input/output data path buffers during process execution.

PCTE allows specification of buffer sizes for "stream" type of input/output by the inherent inclusion of UNIX System V library facilities. However, in general, UNIX drivers perform in-kernel buffering for non-block mode devices (e.g., TTYs). In this case, the user-controlled size request does not affect the driver's internally coded buffering, even though it does modify (tandem) buffering at the level of the UNIX library service routines. Furthermore, many hardware implementations of serial-port interfaces use microcontrollers which provide yet additional levels of buffering, also not under software control. Block mode devices generally are implemented with DMA style transfers, and thus transfer the exact amount of data requested by the user (or the exact sector size where data must be spanned between or within block device *physical sectors).*

6.3F **Datapath Flushing.** The RAC requires that the CAIS shall provide facilities for the removal of all buffered data from an input/output datapath.

PCTE provides support for this function, both for streams, and for read/write (ioctl) interfaces.

6.3G **Output Datapath Processing.** The RAC requires that the CAIS shall provide facilities to force the output of all data in an output datapath.

PCTE provides support for this function, both for streams, and for read/write (ioctl) interfaces.

6.3H **Input/Output Sequencing.** The RAC requires that the CAIS shall provide facilities to ensure the servicing of input/output requests in the order of their invocation.

PCTE support for this requirement is unknown.

6.4 Data Unit Transmission

Section	Conformance	
	requirement	RAC to PCTE
6.4A Data Unit Size	shall provide	Yes
6.4B Raw Input/Output	shall provide	Yes
6.4C Single Data Unit Transmission	shall provide	Yes
6.4D Padding	shall provide	Yes
6.4E Filtering	shall provide	Yes
6.4F Modification	shall provide	Yes
6.4G Input Sampling	shall provide	Yes
6.4H Transmission Characteristics	shall provide	Yes
6.4I Type-Ahead	shall provide	No
6.4J Echoing	shall provide	Yes
6.4K Control Input Datastream	shall provide	Yes
6.4L Control Input Trap	shall provide	Yes
6.4M Trap Sequence	shall provide	Yes
6.4N Data Link Control	shall provide	Yes
6.5A Data Block Size	shall provide	Partial

6.4A **Data Unit Size.** The RAC requires that the CAIS shall provide input/output facilities for communication with devices requiring 1-bit, 7-bit, and 8-bit data units, minimally.

PCTE provides this facility.

6.4B **Raw Input/Output.** The RAC requires that the CAIS shall provide the ability to transmit/receive data units and sequences of units without modification. (Examples of modification are transformation of units, addition of units, and removal of units).

PCTE provides this facility.

6.4C **Single Data Unit Transmission.** The RAC requires that the CAIS shall provide facilities for the input/output of single data units. The completion of this operation makes the data unit available to its consumer(s) without requiring another input/output event, including the receipt of a termination or escape sequence, the filling of a buffer, or the invocation of an operation to force input/output.

PCTE provides this facility.

6.4D **Padding.** The RAC requires that the CAIS shall specify the set of data units and sequences of units (including the null set) which can be added to an input/output datastream. The CAIS shall provide facilities permitting a process to select/query at execution time the subset of data units and sequences of units which may be added (including the null set)

PCTE, at the level of C interfaces, provides support for UNIX's curses and termcap functions. These provide the required capability.

In addition, "ioctl" in PCTE provides control over delay times and fill characters.

6.4E Filtering. The RAC requires that the CAIS shall specify the set of data units and sequences of units (including the null set) which may be filtered from an input or output datastream. The CAIS shall provide facilities permitting a process to select/query at execution time the subset of data units and sequences of units which may be filtered (including the null set).

The PCTE functional description refers "ioctl" (and associated commands) to the UNIX System V description. These features provide the UNIX implementation of these facilities.

In this situation, the PCTE provides a model for meeting the RAC requirements which goes significantly beyond CAIS 1.

6.4F Modification. The RAC requires that the CAIS shall specify the set of modifications that can occur to data units in an input/output datastream (e.g., mapping from lower case to upper case). The CAIS shall provide facilities permitting a process to select/query at execution time the subset of modifications that may occur (including the null set).

For PCTE discussion, see 6.4E.

6.4G Input Sampling. The RAC requires that the CAIS shall provide facilities to sample an input datapath for available data without having to wait if data are not available.

PCTE, by use of underlying UNIX System V facilities, provides this feature.

6.4H Transmission Characteristics. The RAC requires that the CAIS shall support control at execution time of host transmission characteristics (e.g., rates, parity, number of bits, half/full duplex).

PCTE, by use of underlying UNIX System V facilities, provides this feature. Also see 6.4A.

6.4I Type-Ahead. The RAC requires that the CAIS shall provide facilities to disable/enable type-ahead. The CAIS shall provide facilities to indicate whether type-ahead is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable type-ahead in those implementations that do not support type-ahead (e.g., null-effect or exception raised).

PCTE, by its use of the UNIX System V drivers, provides type-ahead, but does not support user control of this function (unless the user were to replace the drivers at run time, which is permitted by System V). Current System V drivers do not give the user control over typeahead, either of its enabling or of its buffering size.

6.4J Echoing. The RAC requires that the CAIS shall provide facilities to disable/enable echoing of data units to their source. The CAIS shall provide facilities to indicate whether echo-suppression is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable echoing in those implementations that do not support echo-suppression (e.g., null effect or exception raised).

For PCTE discussion see 6.4E.

6.4K Control Input Datastream. The RAC requires that the CAIS shall provide facilities to designate an input datastream as a control input datastream

For PCTE discussion, see 6.4E.

6.4L Control Input Trap. The RAC requires that the CAIS shall provide the ability to abort a process by means of trapping a specific data unit or data block in a control input datastream of that process.

For PCTE discussion, see 6.4E.

6.4M Trap Sequence. The RAC requires that the CAIS shall provide facilities to specify/query the data unit or data block that may be trapped. The RAC requires that the CAIS shall provide facilities to disable/enable this facility at execution time.

For PCTE discussion, see 6.4E.

6.4N Data Link Control. The RAC requires that the CAIS shall support facilities for the dynamic control of data links, including, at least, self-test, automatic dialing, hang-up, and broken-link handling.

For PCTE discussion, see 6.4E. (Self-test is not presently defined in the UNIX System V description, but may be implemented by a user-defined driver. In that case, the ioctl command would enable it.)

6.5 Data Block Transmission

6.5A Data Block Size. The RAC requires that the CAIS shall provide facilities for the specification of the size of a sequence of units during program execution.

PCTE provides two kinds of device driver implementations: character mode drivers and block mode drivers. (These are provided by the underlying UNIX System V implementation, and are user replaceable.) Generally, character mode drivers do internal blocking and queuing and the size of a sequence of units is a function of system loading, data rates, memory available, and driver implementor choices. Block mode drivers, on the other hand, are generally implemented such that the block length parameter received from the caller (user) governs the exact length of the transmission block.

For this reason, PCTE compliance is considered partial.

6.6 Data Entity Transfer

Section	Conformance	
	requirement	RAC to PCTE
6.6A Common External Form	shall provide	Yes
6.6B Transfer	shall provide	Yes
6.7A Waiting	shall provide	Ada vendor issue
6.7B Unsupported Features	shall provide	Unknown

6.6A Common External Form. The RAC requires that the CAIS shall specify a representation on physical media of a set of related data entities (referred to as the Common External Form).

PCTE relies on UNIX System V utilities (at least until the PACT toolset provides additional utilities). At this time, the UNIX "tar" and "cpio" functions do define a common external form for data representation on physical media.

In this situation, by virtue of using the standard UNIX utilities, PCTE provides a model for meeting the RAC requirements which goes beyond CAIS 1.

6.6B Transfer. The RAC requires that the CAIS shall provide facilities using the Common External Form to support the transfer among CAIS implementations of sets of related data entities such that attributes and relationships are preserved.

PCTE provides this facility (with caveat noted in 6.6A).

6.7 General Input/Output

6.7A Waiting. The RAC requires that the CAIS shall cause only the task requesting a synchronous input/output operation to await completion.

It is up to the vendor of an Ada compiler (and run time library) to implement a specific model for task operation. It is expected that compiler vendors will provide sub-scheduling, within each process, for the active tasks in the process. While PCTE, by its use of the underlying UNIX model for input/output, supports asynchronous character-device transmission, it does not support asynchronous block mode transmission (e.g., disk input/output). A compiler vendor will need to resort to UNIX's fork operation (or equivalent) to ensure that a process with a task blocked on input/output does not impede the scheduling of other tasks for a given process.

6.7B Unsupported Features. The RAC requires that the CAIS should provide facilities to control the consequences when the physical device does not have all of the features of the virtual device.

PCTE, at the level of C interfaces, provides support for UNIX's CURSES *and* TERMCAP *functions. These provide this capability.*

7. ACKNOWLEDGEMENTS

This document is based on a paper by H. Fischer. We acknowledge contributions from F. Belz in the interpretation of the RAC, from the members of the Ada-Europe Environment Working Group, and from those who commented on earlier drafts. The evaluations expressed here are those of the Ada-Europe Environment Working Group.

July 3, 1986 KITIA Executive Committee Meeting

Attendees: The executive board meeting included B. Abrams, representing Group 1, A. Rudmik, representing Group 2, and D. McGonagle. This meeting was open to KITIA interested parties. The executive board mourns the passing of H. Willman, the chair of Group 3.

A following meeting was held with the above attendees and P. Myers and P. Oberndorf, to discuss the minutes of the Executive Board meeting, and to explore solution avenues to the problems raised

Status of KITIA: The KITIA has been, since January, in a status where its original charter had run out (i.e., members had completing serving the term of membership originally sought). Members had been requesting the AJPO to provide an official statement of invitation to renew their commitment to continue to volunteer service.

P. Myers explained to the general KIT meeting that KITIA's and its relationships with industry. The AJPO would no longer be able to sponsor current mode of operation and sponsorship was in violation of several statutes governing DoD the KITIA. It was explained that KITIA members would be welcome to continue to attend KIT meetings, but would not receive official invitation or be recognized officially as an organization.

The perceived notion by those who heard Myers' speach was that the KITIA was no longer in existence, but the continued contributions by ex-members was sought and desired by the AJPO.

There is significant concern by KITIA members of three areas of concerns: (1) that companies will not continue to send members to government meetings without proper invitation for any number of technicalities, (2) that an association between industry members for the purpose of providing opinions to the government violates anti-lobbying statutes, and (3) that use of government overheads, IR&D funds, and other similar funds to pay for industrial/academic participation to KIT without invitation and sanction may be in violation of statutes.

EXECUTIVE BOARD MEETING

The following items were discussed during the Executive Board Meeting:

TYPES OF RULES WHICH KITIA MIGHT BE VIOLATING

Antitrust or restraint of trade rules: Initial opinion was that KITIA members were not acting in any form of trust for the production of a product which could exclude participation by any segment of industry; to the converse, their assembly was for the purpose of reviewing and influencing a government sponsored product which is an Interface Standard.
Industry-Government lobbying rules: A number of rules govern the

ability of industry members to meet with government members for the purpose of influencing their decisions. KITIA's chartered activities may be in this category.

LEVEL OF INTERACTIONS BETWEEN IND./ACADEMIA AND GOVERNMENT

At government manager level: (This is the level at which the EIA and other "industry organizations" usually operate.) In this form of interaction, government managers (such as Jinny Castor) would provide items for comment to industry and academia, and receive their inputs. Generally there are several layers of government managers between the ones who typically receive industry opinion, and the government individuals who actually execute projects.

At Oberndorf/Contractor level: This is the technical exchange level at which the KITIA has been operating.

DISCUSSION OF SPONSORSHIP OPPORTUNITIES

Self-chartered: This approach basically involves the KITIA's pursuing a course of autonomous operation. It would involve organizational and legal expenses. It would set up the KITIA as an industrial/academic organization for the purpose of technical advice to NOSC (or other TBD agency).

EIA (or other trade association): The EIA has previously expressed a willingness to sponsor the KITIA. There has been extensive discussion on this alternative in about late 1983. The EIA would need to charge an administrative fee (about $300 per year) for representatives from "non-member" organizations. It would bring to bear a strong interface with government managers.

There is concern that an EIA sponsorship might formalize industry/academia interfaces at the government manager level (e.g., via Jinny Castor) and that might prevent direct technical exchange at the lower levels of government interface (e.g., at the Oberndorf/contractor levels).

ACM (or other professional organization): The ACM currently has a group on the CAIS. They have once indicated a willingness to host KITIA activities. They do represent individuals (as technical or other direct contributors), rather than corporations and institutions (as managerial contributors).

There is a concern that the ACM's currently-established strongly negative view toward the CAIS standardization effort might bias the ability of the KITIA to adequately function in their environment in its current role.

LEVEL OF KITIA MEMBER CONTRIBUTION

Individual (representing self): The KITIA members contribute on-the-spot technical opinions. One view is that they act as individuals in making these contributions.

Representing company or academic institution: Most KITIA members have been contributing on a technical level, but influenced by their project, department, or company environment. While contributions are not cleared as "company policy", in nearly all cases they are given with respect to the KITIA member's professional posture in his organization.

FORM OF EXPRESSING VIEWS OF MEMBERS

Need to legitimize inputs: Members feel a strong need for their inputs at KIT meetings to have some sort of official sanction.

Need vehicle to form and express collective view: Members are felt to need a forum for jointly discussing industry/academic viewpoints, and forming collective views.

Need to present collective view at non-managerial government level: Members need the ability to present collective views, and to have the collective views recognized.

DOES CAIS NEED INDUSTRY/ACADEMIA's CONTINUED HELP

Is it on its own course, is influence still needed? This is an open question.

RISKS OF CONTINUED INVOLVEMENT

Association with potential failure: There is some probabilistic possibility that the CAIS will not gain acceptance, either for political or technical reasons. Association with the CAIS at a potential time of failure can be harmful to one's business or career.

QUALITY ASSURANCE GUIDELINES

prepared for

KIT/KITIA

COMPLIANCE WORKING GROUP
(COMPWG)

PRELIMINARY

July 7,1986

by
Lloyd Stiles
Code 844
FCDSSA San Diego

1. Introduction.

Software quality assurance (SQA) is a means for program managers to ensure the development and life-cycle maintenance of high quality computer programs. In large scale embedded computer systems development, the achievment of high quality software systems requires a continuance of comprehensive reviews. These reviews ensure well defined requirements, specifications, design and code. Upon successful completion of each review cycle, the appropriate products are baselined and identified as configuration items (CI). The CIs are placed under configuration control and require formal procedures to implement any changes. The purpose of strong configuration control is to further ensure that the quality built in through top-down engineering and design is not degraded by uncontrolled changes. SQA can be visualized as a management umbrella under which the activities of configuration management (CM) and evaluation and validation (E&V) are carried out. An SQA officer can act as the program manager's coordinator who maintains a system perspective of the entire software development and establishes a system of independent checks and balances in the form of reviews and audits to verify the quality at each phase of development. While it is recognized that excessive SQA activities can kill any project by stifling productivity, too little SQA can allow a poor quality program to be produced.

While not readily accepted by software most engineers, designers and programmers, the implementation of management disciplines into a software project are as important as the engineering disciplines. Figure 2a and 2b illustate the first three to four levels of a software project component structure. Figures 4 and 5 expand the remaining components required for software evaluation and configuration management respectively.

The resources to support software engineering and management varies by project/program and depends on the following factors:

a. The size and complexity of the development effort.

b. The development methodology implemented.

c. The anticipated computer program's life-cycle.

d. The extent of the computer program's visibility and usage.

e. The availability of applicable automated support tools or systems.

f. The requirements and constaints directed by higher authority.

g. The budgetary constaints imposed.

```
Software     << Software      << Resources      << Time
Management    ¦ Development     ¦                 ¦ Funding
              ¦ Planning        ¦                 ¦ Personnel
                                ¦                 ¦ Facilities
                                ¦ Schedules      << PERT
                                                  ¦ CPA
                                                  ¦ Gantt

Quality      << Administration << Quality Assurance Office
Assurance     ¦
              ¦ Evaluation     << Administration << Evaluation and
              ¦                 ¦                 ¦ Validation Office
              ¦                 ¦ Verification   << Document Review
              ¦                 ¦                 ¦ Code Review
              ¦                 ¦                 ¦ ECP/SCP Review
              ¦                 ¦ Monitoring     << Configuration Management
              ¦                                   ¦     Activities
              ¦                                   ¦ Testing Activities
              ¦ Configuration  << Administration << Configuration Management
              ¦ Management      ¦                 ¦    Office
              ¦                 ¦                 ¦ Configuration Control
              ¦                 ¦                 ¦    Boards
              ¦                 ¦ Identification << Baselines
              ¦                 ¦                 ¦ Libraries
              ¦                 ¦                 ¦ Status Accounting
              ¦                 ¦ Control        << Baselines
              ¦                 ¦                 ¦ Libraries
              ¦                 ¦ Status         << Report Generation
              ¦                 ¦   Accounting
              ¦                 ¦
              ¦                 ¦ Audits         << Physical Configuration
              ¦                                   ¦ Functional Configuration
              ¦
              ¦ Validation     << The testing components are listed
                                ¦ in the Validation phase of the
                                ¦ Software Life Cycle
```

Figure 2b. Software Project Component Structure (Management/Quality Assurance)

- Evaluation
  - Administration
    - Evaluation and Validation (E&V) Office
  - Verification (informal, formal, in-progress)
    - Document Review
      - Applicable Evaluation Criteria *
    - Code Review
      - Programming Conventions
      - Error Free Compile
      - Applicable Evaluation Criteria *
    - STR/ECP/SCP Review/Audit
      - System Perspective
      - Applicable Evaluation Criteria *
  - Certification
    - Predevelopment Baselines and Support Software/Tools
      - Physical Configuration Audit (PCA)
      - Functional Configuration Audit (FCA)
  - Program Monitoring
    - Configuration Management Activities
    - Testing Activities

Quality Factors	Evaluation Criteria
Standards	Compliance Technical Editing
Correctness	Traceability Consistency Completeness Feasibility Viability
Integrity (Security)	Access Control Access Audit Process Control
Reliability (Robustness)	Fault Tolerance Consistency Accuracy Simplicity
Efficiency	Execution Efficiency Storage Efficiency
Usability	Training Communicativeness Operability
Flexibility	Modularity Generality Self-descriptiveness Expandability
Maintainability	Consistency Simplicity Conciseness Modularity Self-descriptiveness
Testability	Simplicity Modularity Instrumentation Self-descriptiveness
Portability	Modularity Self-descriptiveness Hardware Independence Software Independence
Reusability	Generality Modularity Hardware Independence Software Independence Self-descriptiveness
Interoperability	Modularity Data Commonality Comunication Commonality
Reproduceability	Code Validity Data Base Validity

Figure 4. Evaluation Component Structure

3. Evaluation Factor Concepts.

The Evaluation components illustrated in Figure 4 provide a list of quailty factors and the corresponding evaluation criteria. This concept is an extension of the quality factors and criteria established in an Air Force study which has been republished by DOD. Table 1 provides definitions for quality factors; Table 2 provides definitions for the related evaluation criteria. While there are many other terms to describe software quality, these tables provide a good definitive list. Implementing quality factors allows a more disciplined approach to software quality assurance. The software program manager (PM) is provided with conceptually simple, easy to use terms for specifying required quality in more precise manner. The PM essentially performs a trade-off analysis for the requirements of the system. The software developers are forced to address how they plan to build the required quality into the software. Specific software quality attributes required are independent of the design and implementation techniques used. Implementation of the applicable quality evaluation criteria provides the PM with a quantifiable criteria against which to judge the software quality prior to acceptance testing and operational use.

In establishing comprehensive reviews it is necessary to go beyond the basic quality factors. Reviews must also include operational and technical evaluation and the conformance to applicable standards. While it usually impossible to find a single person capable of the full comprehensive review of a program, it is possible to have several people review from his or her area of expertise. Thus a subjective review by a single person becomes more objective when reviewed by several people. Further refinement is possible through implementing manual checkoff lists or computer aided reports that define the applicable criteria for each product being reviewed.

4. Summary.

Providing quality software products is accomplished by:

a. Developing product quality through definitive statements of work.

b. Verifying product quality through comprehensive reviews.

c. Validating product quality through thorough testing.

d. Maintaining product quality through stringent configuration control.

Software quality assurance can contribute in the evaluation of the CAIS by providing a definitive set of quality factors that must be prioritized and tailored the CAIS needs. Then the applicable evaluation criteria is expanded into a definitive check-list that can be implemented manually or automatically with computer aided tools.

CRITERIA	DEFINITIONS
Communication Commonality	Those attributes of the software that provide the use of standard protocals and interface routines.
Data Commonality	Those attributes of the software that provide the use of standard data representation.
Modularity	Those attributes of the software that provide a stucture of highly independent modules.
Self-Descriptiveness	Those attributes of the software that provide explanation of the pmplementation of the function.
Hardware Independence	Those attributes of the software that determine its dependency on the hardware system.
Software Independence	Those attributes of the software that determine its dependency on the software environment (operating systems, utilities, input/output routines, etc.)
Compliance	Those attributes of software evaluation that ensure all software components conform to the standards and guidelines imposed.
Technical Editing	Those attributes of software evaluation that ensure all software documents conform to imposed style, format and content standards and are free from spelling and grammarical errors.
Access Control	Those attributes of the software that provide for control of the access of software and data.
Access Audit	Those attributes of the software that provide for an audit of the access of software and data.
Process Control	Those attributes of the software that ensures the security mechanism does not allow unauthorized changes to software or data.
Error Tolerance	Those attributes of the software that provide continuity of operation under adverse operating conditions.
Consistency	Those attributes of the software that provide uniform design and implementation techniques and notation.
Accuracy	Those attributes of the software that provide the required precision in calculations and output.

Table 2. Software Evaluation Criteria Definitions

CRITERIA	DEFINITIONS
Execution Efficiency	Those attributes of the software that provide a minimum processing time.
Storage Efficiency	Those attributes of the software that provide for minimum storage requirements during operation.
Generality	Those attributes of the software that provide breadth to the functions performed.
Expandability	Those attributes of the software that provide for expansion of the data storage requirements or computational functions.
Compliance	Those attributes of software evaluation that ensure all software components (ie: documents, code and reports) conform to the standards and guidelines imposed.
Technical Editing	Those attributes of software evaluation that ensure all software documents conform to imposed style, format and content standards and are free from spelling errors.
Performance	Those attributes of soft·are that support acceptable processing and response time.

Table 2. Software Evaluation Criteria Definitions